**Министерство образования и науки Российской Федерации**
**Федеральное государственное автономное образовательное учреждение**
**высшего образования**
**“Севастопольский государственный университет”**

# Методические указания
# к лабораторным работам по дисциплине
# “Объектно-ориентированное программирование”
# для студентов дневной и заочной форм обучения
# направления 09.03.02 – “Информационные системы и технологии”

**Севастополь**
**2017**

УДК 004.42 (075.8)

Методические указания к лабораторным работам по дисциплине “Объектно-ориентированное программирование” для студентов дневной и заочной форм обучения направления 09.03.02 — “Информационные системы и технологии”/ Сост. Т.И. Сметанина, А.К. Забаштанский — Севастополь: Изд-во СевГУ 2017. — 80с.

Методические указания предназначены для проведения лабораторных работ и обеспечения самостоятельной подготовки студентов по дисциплине “Объектно-ориентированное программирование”. Целью методических указаний является обучение студентов основным принципам объектно-ориентированного программирования, навыкам разработки программ на языке C++.

Методические указания составлены в соответствии с требованиями программы дисциплины “Объектно-ориентированное программирование” для студентов направления 09.03.02 — “Информационные системы и технологии” и утверждены на заседании кафедры информационных систем протокол № __ от _____ 2017 года.

Допущено научно-методической комиссией института информационных технологий и систем управления в технических системах в качестве методических указаний.

Рецензент: Кожаев Е.А., к.т.н., доцент кафедры информационных технологий и компьютерных систем

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

## Цели и задачи лабораторных работ

Основная цель выполнения лабораторных работ — приобретение практических навыков написания программ на объектно-ориентированном языке программирования — C++. В результате выполнения лабораторных работ студенты углубляют знания основных теоретических положений дисциплины «Объектно-ориентированное программирование», решая практические задачи на ЭВМ.

По окончанию курса студенты должны овладеть объектно-ориентированным подходом решения практических задач, овладеть инструментами, сопровождающими разработку программных систем с использованием этого подхода.

## Выбор вариантов и график выполнения лабораторных работ

При изучении курса «Объектно-ориентированное программирование» выполняется 8 лабораторных работ. Каждая работа выполняется в течение 4 академических часов.

В лабораторных работах студент решает заданную индивидуальным вариантом задачу. Варианты заданий приведены в каждой лабораторной работе и уточняются с преподавателем. Номер варианта для студентов заочной формы обучения выбирается в соответствии с номером зачетной книжки студента. Вариант вычисляется как остаток от деления числа, соответствующего двум последним цифрам в номере зачетной книжки, на 15. Например, две последние цифры зачетки 42. Тогда остаток деления 42 на 15 будет равен 12. Следовательно, студент выбирает вариант 12.

Лабораторная работа выполняется в два этапа. На первом этапе, выполняемом самостоятельно дома, студенту необходимо выполнить следующее:

- проработать по конспекту и рекомендованной литературе, приведенной в конце настоящих методических указаний, основные теоретические положения лабораторной работы и ответить на контрольные вопросы;
- разработать алгоритм решения задачи;
- оформить результаты первого этапа в виде заготовки отчета по выполнению лабораторной работы (студенты-заочники первый этап выполнения лабораторных работ оформляют в виде контрольной работы, которую сдают на кафедру до начала зачетно-экзаменационной сессии).

На втором этапе, выполняемом в лабораториях кафедры, студенту следует:

- написать программу на языке Си ++, реализующий разработанный алгоритм;
- отладить программу;
- разработать и выполнить тестовые примеры (не менее двух);
- подготовить и защитить отчёт по лабораторной работе.

Студенты должны выполнять и защищать работы строго по графику.

График выполнения лабораторных работ:

- лабораторная работа №1 — 2-ая неделя осеннего семестра;
- лабораторная работа №2 — 4-ая неделя осеннего семестра;
- лабораторная работа №3 — 6-ая неделя осеннего семестра;
- лабораторная работа №4 — 8-ая неделя осеннего семестра;
- лабораторная работа №5 — 10-ая неделя осеннего семестра;
- лабораторная работа №6 — 12-ая неделя осеннего семестра;
- лабораторная работа №7 — 14-ая неделя осеннего семестра;
- лабораторная работа №8 — 16-ая неделя осеннего семестра.

## Требования к оформлению отчета

Отчёты по лабораторной работе оформляются каждым студентом индивидуально. В общем случае отчёт должен включать: название и номер лабораторной работы; цель работы; вариант задания и постановку задачи; разработку и описание алгоритма решения задачи; разработку и описание программы; выводы по работе; приложения. Содержание отчета приведено в методических указаниях к каждой лабораторной работе.

Постановка задачи представляет изложение содержания задачи, цели её решения. На этом этапе должны быть полностью определены исходные данные, перечислены задачи по преобразованию и обработке входных данных, описаны выходные данные, дана характеристика результатов.

Разработка и описание программы включает описание вычислительного процесса на языке C++. Кроме текста программы, в отчёте представляется её пооператорное описание, даётся характеристика конструкций языка, обеспечивающих решение задачи конкретной лабораторной работы.

В приложении приводится текст программы, результат выполнения тестовых примеров.

# 1 ЛАБОРАТОРНАЯ РАБОТА №1

# ИССЛЕДОВАНИЕ ОСОБЕННОСТЕЙ ОБЪЕКТНОГО ПОДХОДА ПРИ ПРОГРАММИРОВАНИИ СЛОЖНЫХ СТРУКТУР ДАННЫХ С ДИНАМИЧЕСКИМИ ПОЛЯМИ

## 1.1 Цель работы

Исследование основных средств определения класса, создания объектов класса, приобретение навыков разработки и отладки программ, использующих динамическую память. Исследование особенностей использования конструкторов копирования.

## 1.2 Краткие теоретические сведения

### 1.2.1 Описание класса

*Класс* является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними. Данные класса называются *полями* (по аналогии с полями структуры), а функции класса — *методами*. Поля и методы называются элементами класса. Описание класса в первом приближении выглядит так:

```
class <имя>
{
 [ private: ]
<описание скрытых элементов>
[ protected:
<описание защищенных элементов> ]
public:
<описание доступных элементов>
};
```

Описание класса заканчивается точкой с запятой.

Спецификаторы доступа *public, protected* и *private* управляют видимостью элементов класса. Элементы, описанные после служебного слова *private*, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Спецификатор *protected* говорит о том, что данные-члены и функции-члены доступны для функций-членов данного класса и классов, производных от него. Элементы класса, расположенные после спецификатора *public*, являются общедоступными и представляют собой *интерфейс* класса. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций спецификаторов доступа, порядок их следования значения не имеет.

Поля класса:

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

– могут быть описаны с модификатором *const*, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;

– могут быть описаны с модификатором *static*, но не как *auto, extern* или *register.*

Инициализация полей при описании не допускается.

В качестве примера создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупалец, силу или наличие гранатомета) и поведение. Естественно, пример будет схематичен, поскольку приводится лишь для демонстрации синтаксиса.

```
class monstr
{ int health, ammo;
   public:
       monstr(int he = 100, int am = 10){ health = he: ammo = am;}
       void draw(int x, int y, int scale, int position);
       int get_health(){return health;}
       int get_ammo(){return ammo;}
};
```

В этом классе есть два скрытых поля — *health* и *ammo*, получить значения которых извне можно только с помощью методов *get_health()* и *get_ammo().* Доступ к полям с помощью методов в данном случае кажется искусственным усложнением, но надо учитывать, что полями реальных классов могут быть сложные динамические структуры, и получение значений их элементов не так тривиально. Кроме того, очень важной является возможность вносить с эти структуры изменения, не затрагивая интерфейс класса.

Все методы класса имеют непосредственный доступ к его скрытым полям, иными словами, тела функций класса входят в область видимости *private* элементов класса.

В приведенном классе содержится три определения методов и одно объявление (метод *draw*). Если тело метода определено внутри класса, он является встроенным (*inline*). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью **операции расширения области видимости (::)**:

```
void monstr :: draw(int x, int y, int scale, int position){ /* тело метода*/ }
```

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется *конструктором* и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

### 1.2.2 Описание объектов

Конкретные переменные типа “класс” называются экземплярами класса, или объектами. Время жизни и видимость объектов зависит от вида и места их описания. Примеры:

```
monstr Vasia;              // Объект класса monstr с параметрами по умолчанию
monstr Super(200, 300); // Объект с явной инициализацией
monstr stado[100];         // Массив объектов с параметрами по умолчанию
```

```
monstr *beavis = new monstr (10); // Динамический объект
                                  // (второй параметр задается по умолчанию)
monstr &butthead = Vasia;         // Ссылка на объект
```

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

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция "**.**" (точка) при обращении к элементу через имя объекта и операция "**–>**" при обращении через указатель, например:

```
int n = Vasia.get_ammo();
stado[5].draw(2,3,4,5);
cout << beavis–>get_health();
```

Обратиться таким образом можно только к элементам со спецификатором *public*. Получить или изменить значения элементов со спецификатором *private* можно только через обращение к соответствующим методам, либо невозможно вовсе, если таких методов в классе нет.

### 1.2.3 Конструкторы

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов:

1. Конструктор не возвращает значение, даже типа *void*. Нельзя получить указатель на конструктор.

2. Конструкторы нельзя описывать с модификаторами *const, virtual u static*.

3. Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

4. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки). Параметры конструктора могут иметь любой тип, кроме этого же класса.

5. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

6. Если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.

7. Конструкторы глобальных объектов вызываются до вызова функции *main*. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

8. Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

*имя_класса имя_объекта [(список параметров)];*
// Список параметров не должен быть пустым
*имя_класса (список параметров);*
// Создается объект без имени (список может быть пустым)
*имя_класса имя_объекта = выражение;*
// Создается объект без имени и копируется

Примеры:

```
monstr Super(200, 300), Vasia(50), Z;
monstr X = monstr(1000);
monstr Y = 500;
```

В первом операторе создаются три объекта. Значения не указанных параметров устанавливаются по умолчанию. Во втором операторе создается безымянный объект со значением параметра *health = 1000* (значение второго параметра устанавливается по умолчанию). Выделяется память под объект *X*, в которую копируется безымянный объект. В последнем операторе создается безымянный объект со значением параметра *health = 500* (значение второго параметра устанавливается по умолчанию). Выделяется память под объект *Y,* в которую копируется безымянный объект. Такая форма создания объекта возможна в том случае, если для инициализации объекта допускается задать один параметр.

В качестве примера класса с несколькими конструкторами усовершенствуем описанный ранее класс *monstr*, добавив в него поля, задающие цвет (*skin*) и имя (*name*):

```
enum color {red, green, blue}; // Возможные значения цвета
class monstr
{
  int health, ammo;
  color skin;
  char *name;

public:
  monstr(int he = 100, int am =10);
  monstr(color sk);
  monstr(char * nam);
  int get_health(){return health;}
  int get_ammo(){return ammo;}
};
//------------------------
monstr::monstr(int he, int am)
{ health = he; ammo = am; skin = red; name = 0;
}
// ----- --------------------
monstr::monstr(color sk)
{
   switch (sk)
```

```
  { case red : health = 100; ammo =10; skin = red; name = 0; break:
      case green: health = 100; ammo = 20; skin = green; name = 0; break;
      case blue : health = 100; ammo = 40; skin = blue; name = 0; break;
  }
}
// ------------------------
monstr::monstr(char * nam)
{       name = new char [strlen(nam) + 1];
        // К длине строки добавляется 1 для хранения нуль-символа
        strcpy(name, nam);
        health = 100; ammo = 10; skin = red;
}
// ------------------------
monstr * m = new monstr («Ork»);
monstr Green (green);
```

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров. Объекты класса *monstr* теперь можно инициализировать различными способами, требуемый конструктор будет вызван в зависимости от списка значений в скобках. При задании нескольких конструкторов следует соблюдать те же правила, что и при написании перегруженных функций — у компилятора должна быть возможность распознать нужный вариант.

Существует еще один способ инициализации полей в конструкторе:

```
monstr::monstr(int he, int am):
health (he), ammo (am), skin (red), name (0){}
```

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации полей-констант, полей-ссылок и полей-объектов. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

### 2.4 Конструктор копирования

Помимо стандартных конструкторов, часто используется конструктор копирования, который вызывается всякий раз, когда необходимо создать копию объекта.

При передаче и возвращении объекта в функцию в виде значения всегда создается его временная копия. Если объект является пользовательским, то вызывается конструктор копирования его класса.

Все конструкторы копирования имеют только один параметр — ссылку на объект своего класса. Разумно сделать эту ссылку постоянной, поскольку конструктор не должен изменять передаваемый ему объект. Например:

```
monstr (const monstr & theCat);
```

В данном случае конструктор *monstr* принимает постоянную ссылку на объект класса *monstr*, ведь задачей конструктора является создание копии *theCat*.

Стандартный конструктор копирования (создаваемый компилятором по умолчанию) просто копирует каждую переменную переданного ему объекта в переменные нового объекта. Такое копирование называется ***поверхностным*** (*shallow copy*). Хоть оно и подходит для большинства случаев, могут возникнуть серьезные проблемы, если в классе поля являются указателями на объекты в динамической памяти.

В результате поверхностного копирования создаются точные копии значений всех полей одного объекта в другом. Следовательно, указатели в обоих объектах будут указывать на одну и ту же область динамической памяти.

Катастрофа случится тогда, когда любой из объектов прекратит свое существование. Как уже говорилось, задача деструктора состоит в освобождении памяти, занимаемой объектом. Если деструктор первого объекта освободит динамическую память, а второй объект будет указывать на нее, то возникнет паразитный указатель, а работа программы не будет стабильной и предсказуемой.

Во избежание подобных проблем, необходимо вместо стандартного конструктора копирования использовать собственный, который будет осуществлять ***глубокое копирование*** с перемещением значений полей, находящихся в динамической памяти. Глубокое копирование переносит значения, находящиеся в динамической памяти, во вновь созданные участки.

### 2.5 Деструкторы

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

– для локальных объектов — при выходе из блока, в котором они объявлены;

– для глобальных — как часть процедуры выхода из *main*;

– для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции *delete.*

Деструктор должен выглядеть так:

```
monstr::~monstr() {delete [] name;}
```

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса. Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор. Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически — иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как *const* или *static*;
- не наследуется, но может быть виртуальным.

### 2.6 Сборщик мусора

При работе с динамическими объектами программист должен прописывать выделение памяти с помощью операции *new*, следить за используемостью и достижимостью динамических объектов и, по необходимости, вручную же очищать занимаемую ими память с помощью вызова операции *delete*.

При этом возможны такие проблемы:

1. Висячая ссылка — это оставшаяся в использовании ссылка на объект, который уже удалён. После удаления объекта все сохранившиеся в программе ссылки на него становятся «висячими». Попытка обратиться по такой ссылке приводит к непредсказуемым последствиям или аварийной остановке программы.

2. Утечка памяти — это ситуация, когда объект в динамической памяти уже не используется, но не был удален программистом, а ссылающейся на него переменной было присвоено новое значение. В таком случае, если на объект нет других ссылок, он становится программно недоступным, но продолжает занимать память, освободить которую не представляется возможным.

Если в программе постоянно создаются и «теряются» объекты, то из-за утечек памяти (при долгой работе программы) возможно существенное замедление работы системы или завершение приложения с ошибкой из-за нехватки свободной памяти.

Сборщик мусора (Gargabe Collector, GC) — это способ автоматического управления оперативной памятью, выделяемой в куче. Суть его работы сводится к тому, что программист выделяет динамическую память под необходимые ему объекты, а сборщик мусора уже сам определит, когда они станут не нужны, и освободит память. Это решает большинство проблем с утечками памяти, однако во многих случаях системы со сборкой мусора демонстрируют меньшую эффективность, как по скорости работы, так и по объёму используемой памяти. С другой стороны, сборщик мусора существенно упрощает процесс программирования и делает его более безопасным, защищая от ошибок в использовании динамической памяти.

В случае использования данного метода при программировании в классах нет необходимости описывать деструкторы, так как удаление будет выполнено автоматически, как объекта, так и связанных с ним динамических структур данных. Поэтому в средах программирования, реализующих сборку мусора, зачастую нет понятия деструктора класса.

### 1.3 Порядок выполнения лабораторной работы

1.3.1 В ходе самостоятельной подготовки изучить основные средства языка C++ для определения класса, создания объектов класса, работы с такими объектами и их уничтожения после использования.

1.3.2 Разработать программу на языке C++. Работа программы должна быть реализована в виде меню со следующими пунктами:

- создание объекта;
- создание копии объекта (используя конструктор копирования);
- просмотр полей оригинала и копии;

- изменение одного из полей объекта (копии или оригинала);
- выполнение задания согласно варианту.

1.3.3 Разработать тестовые примеры.

1.3.4 Выполнить отладку программы.

1.3.5 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы, проанализировав преимущества и недостатки структурного и объектного подхода в применении к этой задаче.

1.3.6 Оформить отчет по проделанной работе.

### 1.4 Варианты заданий

Необходимо для заданного по варианту динамического типа данных описать класс, содержащий указатель на динамический тип как поле данных. Для этого класса описать конструкторы (не менее трех, в том числе и конструктор копирования), деструктор, функцию печати данных. Создать экземпляр полученного класса и проиллюстрировать его корректную работу: распечатать данные, изменить данные и распечатать вновь. Создать второй экземпляр класса как копию первого и проиллюстрировать корректную работу конструктора копирования: распечатать и изменить данные объекта-копии, распечатать данные обоих объектов, сравнить результат. Предусмотреть обработку ошибок при манипуляции с данными.

**Вариант 1**

Динамическая структура — очередь. В списке хранится информация о событиях: день (1-31), месяц (1-12), год (целое число) и название (строка). Предусмотреть функции добавления элементов в список и удаления из него, а также поиска введенной даты.

**Вариант 2**

Динамическая структура — стек. В стеке хранится распорядок дня: час (0-23), минуты (0-59), задача (строка). Предусмотреть функции удаления элементов из стека и добавления нового элемента в стек с сохранением упорядоченности по времени.

**Вариант 3**

Динамическая структура — очередь. В списке хранится информация о чеках: наименование товара (строка), цена товара и дата продажи. Предусмотреть функции добавления элементов в список и удаления из него, а также поиска товаров, цена которых превышает значение, вводимое пользователем.

**Вариант 4**

Динамическая структура — циклическая очередь. В ней хранятся данные о возрасте: ФИО (строка) и возраст (число). Предусмотреть функции добавления элементов в очередь и удаления из нее, а также функцию поиска по фамилии.

**Вариант 5**

Динамическая структура — двусвязный список. Хранит информацию о товарах на складе: инвентарный номер (число 5 знаков), название товара (строка). Предусмот-

реть функции добавления элементов в список и удаления из него, а также функцию проверки, существует ли товар на складе, по инвентарному номеру.

**Вариант 6**

Динамическая структура — очередь. Хранит данные о химических элементах: количество протонов в ядре (число 1-200) и название химического элемента (строка). Предусмотреть функции добавления элементов в очередь и удаления из нее, а также проверку на вхождение в список элементов с одинаковым значением заряда ядра.

**Вариант 7**

Динамическая структура — двусвязный список. Хранит анкетные данные студентов: ФИО (строка) и дата рождения. Предусмотреть функции добавления элементов в список и удаления из него, а также функцию проверки, существует ли в списке студенты, родившиеся после даты, вводимой пользователем.

**Вариант 8**

Динамическая структура — циклическая очередь. Хранит информацию о штрафах: номер автомобиля (строка символов 8-9) и величина штрафа (цифра). Предусмотреть функции добавления элементов в очередь и удаления из нее, а также функцию вычисления величины суммы штрафов со всех авто.

**Вариант 9**

Динамическая структура — очередь. Хранит библиотечные данные: автора книги (строка) и название книги (строка). Предусмотреть функции добавления элементов в очередь и удаления из нее, а также функцию поиска всех произведений введенного автора.

**Вариант 10**

Динамическая структура — стек. В стеке хранится сводная ведомость оценок студентов: ФИО (строка) и пять оценок. Предусмотреть функции удаления элементов из стека и добавления нового элемента в стек, а также функцию поиска среднего балла.

**Вариант 11**

Динамическая структура — двусвязный список. Хранимые данные — поставки железной руды на плавильную печь: номер поставки (число), вес руды (число) и ожидаемый выход металла (число 0.0-0.9). Предусмотреть функции добавления элементов в список и удаления из него, а также функцию поиска суммарного веса чистого металла.

**Вариант 12**

Динамическая структура — циклическая очередь. Хранимая информация — каталог монет: порядковый номер (число), название монеты (строка), материал изготовления (строка). Предусмотреть функции добавления элементов в очередь и удаления из нее, а также функцию подсчета количества видов монет в каталоге из заданного материала.

**Вариант 13**

Динамическая структура — стек. В стеке хранится сводная ведомость оценок студентов: ФИО (строка) и пять оценок. Предусмотреть функции удаления элементов из стека и добавления нового элемента в стек, а также функцию поиска фамилий студентов, средний балл которых выше 4.0.

**Вариант 14**

Динамическая структура — очередь. Хранится информация об аукционе: номер товара (число), название товара (строка), начальная стоимость товара (число). Предусмотреть функции добавления элементов в очередь и удаления из нее, а также функцию подсчета ожидаемой прибыли от проведения аукциона.

**Вариант 15**

Динамическая структура — стек. В стеке хранится сводная ведомость оценок студентов: ФИО (строка) и пять оценок. Предусмотреть функции удаления элементов из стека и добавления нового элемента в стек, а также функцию поиска фамилий студентов, имеющих хотя бы одну тройку.

### 1.5 Содержание отчета

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 1.6 Контрольные вопросы

1.6.1 Объясните понятие “класс”.
1.6.2 Объясните понятия “данные-члены” и “функции-члены” класса.
1.6.3 Объясните применение операция расширения области видимости.
1.6.4 Объясните понятие “инкапсуляция”.
1.6.5 Поясните значение директив *public*, *protected* и *private*.
1.6.6 Опишите как осуществляется доступ к членам класса.
1.6.7 Объясните понятие “конструктор класса” и его применение.
1.6.8 Объясните понятие “конструктор по умолчанию”.
1.6.9 Перечислите свойства конструкторов.
1.6.10 Объясните понятие “конструктор копирования” и его применение.
1.6.11 Объясните понятие “деструктор класса”.
1.6.12 Перечислите свойства деструкторов.
1.6.13 Объясните понятие “механизм утечек памяти” и причины возникновения.
1.6.14 Объясните понятие “сборщик мусора” и его применение.

# 2 ЛАБОРАТОРНАЯ РАБОТА № 2

# ИССЛЕДОВАНИЕ МЕХАНИЗМА ВИРТУАЛЬНЫХ ФУНКЦИЙ

## 2.1 Цель работы

Приобретение практических навыков при написании объектно-ориентированных программ с использованием механизмов наследования и виртуальных функций. Освоение особенностей отладки объектно-ориентированных программ.

## 2.2 Краткие теоретические сведения

### 2.1 Простое наследование

Язык C++ позволяет классу наследовать данные-члены и функции-члены одного или нескольких других классов. При этом новый класс называют ***производным*** классом (или классом-потомком). Класс, элементы которого наследуются, называется ***базовым*** классом (родительским классом или классом-предком) для своего производного класса. Наследование дает возможность некоторые общие черты поведения классов абстрагировать в одном базовом классе. Производные классы, наследуя это общее поведение, могут его несколько видоизменять, переопределяя некоторые функции-члены базового класса, или дополнять, вводя новые данные-члены и функции-члены. Таким образом, определение производного класса значительно сокращается, поскольку нужно определить только отличающие его от производных классов черты поведения.

Если у производного класса имеется всего один базовый класс, то говорят о ***простом*** (или одиночном) ***наследовании***. Синтаксис объявления производного класса имеет следующий вид:

```
class <имя класса>:[<спецификатор доступа>] <имя базового класса>
{
    <тело класса>
};
```

***Спецификатор доступа*** — это одно из ключевых слов *public*, *protected* или *private*. Он не является обязательным и по умолчанию принимает значение *private* для классов и *public* для структур. Рассмотрим пример:

```
class Base  // Базовый класс
{  public:
   int SetX(int _x=0) { return x=_x;}
   int GetX( ) {return x;}
protected:                    // Обозначает, что данное поле доступно
   int x;                     // данному классу и всем его потомкам
};

class Derived : public Base  // Класс-наследник
{      };
```

```
int main(void)          // Основная программа
{       int x;
        Derived ob;     // Создаем объект класса-наследника
        x = ob.GetX( ); // и обращаемся в нем к полям и методам базового класса
        return 0;
}
```

Спецификатор доступа при наследовании определяет уровень доступа к элементам базового класса, который получают элементы производного класса. В приведенной ниже таблице описаны возможные варианты наследуемого доступа.

Таблица 2.1 — Распространение спецификаторов доступа в рамках иерархии наследования

| Доступ в базовом классе | Спецификатор наследуемого доступа | Доступ в производном классе |
|---|---|---|
| *Public*<br>*Protected*<br>*Private* | *Public* | *Public*<br>*Protected*<br>Нет доступа |
| *Public*<br>*Protected*<br>*Private* | *Protected* | *Protected*<br>*Protected*<br>Нет доступа |
| *Public*<br>*Protected*<br>*Private* | *Private* | *Private*<br>*Private*<br>Нет доступа |

Можно сделать вывод, что спецификатор наследуемого доступа устанавливает тот уровень доступа, до которого понижается уровень доступа к элементам, установленный в базовом классе.

Следует отметить, что не все члены класса будут наследоваться. Не подлежат наследованию следующие элементы класса:

- конструкторы;
- конструкторы копирования;
- деструкторы;
- операторы присваивания, определенные программистом;
- друзья класса.

## 2.2 Переопределение функций

В производном классе обычно добавляются новые поля и методы, не существующие в базовом классе. Однако существует также возможность переопределения функций-членов базового класса. Чтобы переопределить функцию-член базового класса в производном классе, достаточно включить ее прототип в объявление этого класса и затем дать ее определение. Конечно, прототипы переопределяемой функции в базовом и производном классах должны совпадать. Рассмотрим пример:

```
#include <iostream.h>
class MyInt
{ protected:
        int x;
 public:
      MyInt(int _x) {x = _x;}
      MyInt( ){x = 0;}
      int GetX( ){return x;}    // Определим метод вывода поля для базового класса
      void SetX(int _x) {x = _x;}
};
class IncMyInt: public MyInt
{ public:
      IncMyInt(int _x) : MyInt(_x) {}
      int GetX( ) {return ++x;} // Переопределим для наследника метод
                                // базового класса
      void ShowX( ) {cout << GetX( )<<' ';}
};
main( )
{
      IncMyInt obj(10);          // Определение объекта класса

      obj.ShowX( );              // Вызов функции класса-наследника
      return 0;
}
```

В этом случае функция-член *ShowX()* объекта *obj* использует для получения значений данных-членов переопределенный вариант функции базового класса *GetX()*. В результате на экран программа выведет инкрементированное значение заданного числа.

Иногда возникает необходимость вызвать функцию-член базового класса, а не ее переопределенный вариант. Это можно сделать с помощью операции ***расширения области видимости***, применяемой в форме:

< имя_класса> **::** <имя_функции>

Данная операция позволяет компилятору "видеть" за пределами текущей области видимости. Особенно часто это приходится делать при переопределении функций-членов. Переопределяемая функция-член может вызывать соответствующую функцию-член базового класса, а затем выполнять некоторую дополнительную работу (или наоборот). Например, в предыдущем примере можно было переопределить функцию *GetX()* следующим образом:

```
int IncMyInt:: GetX( )          // Переопределение метода в наследнике
{
    int z ;
```

```
    z = MyInt::GetX( ) ;        // Вызов функции из базового класса в
                                // описании метода в классе-наследнике
    return ++z;
}
```

## 2.3 Конструкторы и деструкторы

Конструкторы не наследуются, поэтому производный класс либо должен объявить свой конструктор, либо предоставить возможность компилятору генерировать конструктор по умолчанию. Поскольку производный класс должен унаследовать все члены родительского, при построении объекта своего класса он должен обеспечить инициализацию унаследованных данных-членов, причем она должна быть выполнена до инициализации данных-членов производного класса, так как последние могут использовать значения первых. В связи с этим в производном классе применяется следующая конструкция:

<имя класса>([<список аргументов>]) **:** <имя класса предка>( [<список аргументов>])

То есть используется список инициализации элементов, в котором указывается конструктор базового класса. Часть параметров, переданных конструктору производного класса, обычно используется в качестве аргументов конструктора базового класса. Затем в теле конструктора производного класса выполняется инициализация данных-членов, принадлежащих собственно этому классу.

В приведенном выше примере эта конструкция использована для создания конструктора класса *IncMyInt*. В данном случае конструктор имеет вид:

```
IncMyInt (int_x) : MyInt (_x) {}
```

Тело конструктора оставлено пустым, так как класс *IncMyInt* не имеет собственных данных-членов. Все параметры конструктора производного класса просто переданы конструктору базового класса для осуществления инициализации.

Если предоставляемый программистом конструктор не имеет параметров, то есть является конструктором по умолчанию, при создании экземпляра производного класса автоматически вызывается конструктор базового класса. После того как объект создан, конструктор базового класса становится недоступным.

В отношении деструкторов производных классов также действуют определенные правила. Деструктор производного класса должен выполняться раньше деструктора базового класса (иначе деструктор базового класса мог бы разрушить данные-члены, которые используются и в производном классе). Когда деструктор производного класса выполнит свою часть работы по уничтожению объекта, вызывается деструктор базового класса. Причем всю работу по организации соответствующих вызовов выполняет компилятор, программист не должен заботиться об этом.

## 2.4 Виртуальные методы

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса, например:

```
MyInt *p; // Описывается указатель на базовый класс
p = new IncMyInt; // Указатель ссылается на объект производного класса
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например,

```
p->SetX(123);
```

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

```
(IncMyInt * p)-> SetX (123);
```

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример — связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В C++ реализован механизм ***позднего связывания,*** когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов.

Функции, у которых известен интерфейс вызова (то есть прототип), но реализация не может быть задана в общем случае, а может быть определена только для конкретных случаев, называются ***виртуальными*** (термин, означающий, что функция может быть переопределена в производном классе).

Механизм виртуальных функций гарантирует, что для объекта будет вызнана правильная функция независимо от того, какое выражение используется для осуществления вызова.

Для определения ***виртуального метода*** используется спецификатор *virtual*, например:

```
virtual void draw(int x, int y, int scale, int position);
```

Рассмотрим правила описания и использования виртуальных методов:

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

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

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

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

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

Итак, ***виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы*** (перевод красивого английского слова *virtual* — в данном значении всего-навсего “фактический”, то есть ссылка разрешается по факту вызова).

## 2.5 Механизм позднего связывания

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает ***таблицу виртуальных методов*** (*vtbl*), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в *vtbl* одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное ***поле ссылки*** на *vtbl*, называемое *vptr*. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к *vtbl* через *vptr* объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Поскольку объекты с виртуальными функциями должны поддерживать и таблицу виртуальных функций, то их использование всегда ведет к некоторому повышению затрат памяти и снижению быстродействия программы. Если вы работаете с небольшим классом, который не собираетесь делать базовым для других классов, то в этом случае нет никакого смысла в использовании виртуальных функций.

## 2.6 Виртуальный деструктор

***Рекомендуется делать виртуальными деструкторы*** для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции delete размер объекта, имеющий тип *size_t*. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, передаваемый размер объекта может оказаться неправильным.

При удалении объекта производного класса, на который ссылается указатель базового класса, если деструктор объявлен как виртуальный, то будет вызван деструктор соответствующего производного класса. Затем деструктор производного класса вызовет деструктор базового класса, и объект будет правильно удален (удален целиком). В противном случае будет вызван только деструктор базового класса и произойдет утечка памяти. Рассмотрим пример:

```
class Base
{       int x;
`public:
        Base(_x){x = _x;};
        ~Base( )  { /*Освобождение памяти */ }
};
class Derived: public Base
{ public:
       ~Derived{ )  { /*Освобождение памяти */ }
};
int main ( )
{       Base* pBase = new Derived;
        //...
        delete pBase;
        return 0;
}
```

Так как объявленный в классе *Derived* деструктор не является виртуальным, инструкция "*delete pBase;*" вызовет удаление только памяти, принадлежащей классу *Base*, поскольку она будет квалифицирована как вызов *Base::~Base()*. Это может привести к утечке памяти. Если же сделать деструктор базового класса виртуальным, то и деструктор производного класса будет виртуальным. Значит использование предыдущей инструкции будет квалифицировано как вызов "*Derived :: ~ Derived();*" и удаление объекта произойдет правильно.

В связи с этим можно сформулировать следующие правила:

- используйте виртуальный деструктор, если в базовом классе имеются виртуальные функции;

- используйте виртуальные функции только в том случае, если программа содержит и базовый, и производный классы;

- не пытайтесь создать виртуальный конструктор.

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

## 2.7 Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется ***абстрактный***.

***Чисто виртуальный метод*** — это метод класса, не имеющий описания. Чисто виртуальный метод содержит признак "= 0" вместо тела, например:

```
virtual void func (int) = 0;
```

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

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

При определении абстрактного класса необходимо иметь в виду следующее:

a) абстрактный класс не может использоваться в качестве типа аргумента, передаваемого функции;
b) абстрактный класс не может использоваться в качестве типа возвращаемого значения функции;
c) нельзя осуществлять явное преобразование типа объекта к типу абстрактного класса;
d) нельзя объявить представитель абстрактного класса;
e) можно объявить указатель или ссылку на абстрактный класс.

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

### 2.3 Порядок выполнения лабораторной работы

2.3.1 В ходе самостоятельной подготовки изучить основы работы с механизмами наследования и виртуальных функций.

2.3.2 Разработать программу на языке C++, реализующую следующие действия:

- создание объектов производных классов;
- заполнение полей созданных объектов;
- вызов виртуальных функций.

2.3.3 Разработать тестовые примеры и выполнить отладку программы.

2.3.4 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

2.3.5 Оформить отчет по проделанной работе.

### 2.4 Варианты заданий

Для заданной по варианту иерархии описать классы, конструкторы и деструктор, функции ввода и вывода информации на экран. Базовый класс определить как абстрактный, а заданную функцию — как чисто виртуальную в базовом классе и переопределить ее в остальных классах иерархии. Проиллюстрировать корректную работу виртуальных функций и механизма наследования.

***Вариант 1***

Создать абстрактный базовый класс Фигура с виртуальной функцией Площадь. Создать производные классы: прямоугольник, круг, прямоугольный треугольник со своими функциями площади.

***Вариант 2***

Создать абстрактный базовый класс с виртуальной функцией: Норма. Создать производные классы: комплексные числа, вектор из 10 элементов, матрица (2x2). Определить функцию нормы: для комплексных чисел — модуль в квадрате, для вектора — корень квадратный из суммы элементов по модулю, для матрицы — максимальное значение по модулю.

***Вариант 3***

Создать абстрактный базовый класс Фигура и производные классы: круг, прямоугольник, четырехугольник. Определить виртуальную функцию вычисления периметра.

***Вариант 4***

Создать абстрактный базовый класс Вектор (одномерный массив) с виртуальной функцией Сумма, вычисляющей сумму элементов вектора. Создать производные классы: вектор целых чисел (*int**) и два вектора вещественных чисел (*float** и *double**).

***Вариант 5***

Создать абстрактный базовый класс Прогрессия с виртуальной функцией Сумма прогрессии. Создать производные классы: арифметическая прогрессия и геометрическая прогрессия. Каждый класс имеет два поля типа *double*. Первое — первый член прогрессии, второе — постоянная разность (для арифметической) и постоянное отношение (для геометрической). Определить функцию вычисления суммы, где параметром является количество элементов прогрессии.

Арифметическая прогрессия $a_j = a_0 + jd, j = 0, 1, 2, \ldots$

Сумма арифметической прогрессии: $s_n = (n+1)(a_0 + a_n) / 2$

Геометрическая прогрессия: $a_j = a_0 r^j$, $j = 0, 1, 2, \ldots$

Сумма геометрической прогрессии: $s_n = (a_0 - a_n r) / (1-r)$

***Вариант 6***

Создать абстрактный базовый класс Матрица с виртуальной функцией поиска максимального значения в массиве. Создать производные классы: матрица целых чисел (*int***), матрица символов (*char* **) и матрица вещественных чисел (*double***).

***Вариант 7***

Создать абстрактный базовый класс Уравнение с виртуальной функцией нахождения значения *y* по заданному значению *x*. Создать производные классы: линейное уравнение, квадратное уравнение и кубическая парабола.

***Вариант 8***

Создать абстрактный базовый класс Число с виртуальной функцией изменения знака числа. Создать производные классы: целое (*int*), вещественное (*double*) и комплексное (*float, float*).

***Вариант 9***

Создать абстрактный базовый класс Фигура с виртуальной функцией Объем. Создать производные классы: параллелепипед, тетраэдр, шар со своими функциями объема. Объем параллелепипеда вычисляется по формуле $V = xyz$ ($x$, $y$, $z$ — стороны), тетраэдра: $V = \frac{\sqrt{2}}{12} a^3$, шара: $V = \frac{4}{3} \pi R^3$

***Вариант 10***

Создать абстрактный базовый класс Вектор (одномерный массив) с виртуальной функцией Сортировка, которая упорядочивает вектор по возрастанию значения элементов. Создать производные классы: вектор целых чисел (*int**) и два вектора вещественных чисел (*float** и *double**).

***Вариант 11***

Создать абстрактный базовый класс Поиск с виртуальной функцией поиска. Создать производные классы: символ (*char*), строка (*char* *) и матрица (*char* **). Функция проверяет, существует ли введенная буква в строке/матрице/равна ли символу?

***Вариант 12***

Создать абстрактный базовый класс Студент с виртуальной функцией печати информации. Создать производные классы: студент-дипломник (тема диплома, специализация как бакалавра), студент-бакалавр (тема работы), студент младших курсов.

***Вариант 13***

Создать абстрактный базовый класс Матрица с виртуальной функцией поиска наиболее часто встречаемого элемента. Создать производные классы: матрица целых чисел (*int***), матрица символов (*char***) и матрица вещественных чисел (*double***).

***Вариант 14***

Создать абстрактный базовый класс Произведение, у которого есть виртуальная функция печати данных. Создать производные классы: книга (поля: автор, название, жанр, рег. номер) и картина (поля: автор, название, стиль, номер, размер).

***Вариант 15***

Создать абстрактный базовый класс Человек, у которого есть виртуальная функция печати личных данных. Создать производные классы: мужчина (поля: ФИО, возраст, должность, статус, имя жены) и женщина (поля: ФИО, возраст, статус, имя мужа, дети, имена детей).

### 2.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

## 2.6 Контрольные вопросы

2.6.1 Что такое простое наследование, для чего оно применяется?

2.6.2 Дайте определения следующим терминам: базовый класс, производный класс.

2.6.3 Что такое спецификатор доступа, для чего он нужен при наследовании?

2.6.4 Все ли элементы класса можно наследовать? Какие нельзя?

2.6.5 Привести пример переопределения функции и объяснить суть механизма.

2.6.6 Как обратиться к методу базового класса, если он был переопределен в наследнике?

2.6.7 Перечислить особенности описания и поведения конструкторов и деструкторов при наследовании.

2.6.8 Пояснить работу механизмов раннего и позднего связывания.

2.6.9 Дать определение термина виртуальный метод. Привести пример. Перечислить основные правила по работе с виртуальными методами.

2.6.10 Описать содержимое и принцип работы таблицы виртуальных методов.

2.6.11 Пояснить необходимость создания в программе виртуальных деструкторов. Приведите пример.

2.6.12 Дать определение термина полиморфизм.

2.6.13 Абстрактный класс — определение, назначение, пример описания.

2.6.14 Перечислить правила работы с абстрактным классом.

# 3 ЛАБОРАТОРНАЯ РАБОТА № 3

# ИССЛЕДОВАНИЕ МЕХАНИЗМА МНОЖЕСТВЕННОГО НАСЛЕДОВАНИЯ

## 3.1 Цель работы

Приобретение практических навыков при написании объектно-ориентированных программ с использованием механизма множественного наследования.

## 3.2 Краткие теоретические сведения

### 3.2.1 Множественное наследование

Если у производного класса имеется несколько базовых классов, то говорят о множественном наследовании. Множественное наследование позволяет сочетать в одном производном классе свойства и поведение нескольких классов.

```
#include < iostream.h>
#include < conio.h>
#include < string.h>
class Coord
{  int x, y;
public:
        Coord(int _x, int _y) {x = _x; y = _y;}
        int GetX( ){return x;}
        int GetY( ) {return y;}
        void SetX(int _x){x = _x;}
        void SetY(int _y){y = _y;}
};

class SaveMsg
{   char Message[80];
public:
       SaveMsg(char* msg){SetMsg(msg);}
        void SetMsg(char* msg) {strcpy(Message, msg);}
        void ShowMsg() {cout << Message;}
};

class PrintMsg: public Coord, public SaveMsg
{public:
       PrintMsg (int _x, int _y, char* msg) : Coord(_x, _y), SaveMsg(msg){ };
       void Show();
};

void PrintMsg::Show()
{        gotoxy(GetX( ),GetY( ));
        Show*Msg ();
```

```
        }
main( )
{       PrintMsg * ptr;
        ptr =new PrintMsg(10, 5, "Множественное");
        ptr->Show( ) ;
        ptr->SetX(25);
        ptr->SetY(5);
        ptr->SetMsg("наследование");
        ptr->Show() ;
        delete ptr;
        return 0;
}
```

В этом примере класс *Coord* отвечает за хранение и установку значений координат на экране, класс *SaveMsg* хранит и устанавливает сообщение, а класс *PrintMsg*, являющийся производным от них, выводит это сообщение на экран в заданных координатах. Выполнение функции-члена *Show()* приводит к выводу на экран в указанных координатах сообщения, заданного третьим аргументом. Следующие за этим вызовы функций-членов *SetX()*, *SetY()* и *SetMsg()* приводят к установке новых значений координат и нового сообщения, которое выводится повторным вызовом функции *Show()*.

Этот пример демонстрирует также и то, как осуществляется передача параметров конструкторам базовых классов. При множественном наследовании, как и при простом, конструкторы базовых классов вызываются компилятором до вызова конструктора производного класса. Единственная возможность передать им аргументы – использовать список инициализации элементов. Причем порядок объявления базовых классов при наследовании определяет и порядок вызова их конструкторов, которому должен соответствовать порядок следования конструкторов базовых классов в списке инициализации. В данном случае класс *PrintMsg* содержит такое объявление о наследовании:

```
class PrintMsg: public Coord, public SaveMsg
```

Этому объявлению соответствует следующий конструктор:

```
PrintMsg (int _x, int __y, char* msg) : Coord(_x, _y), SaveMsg(msg){}
```

В основной программе этому конструктору передаются аргументы при создании объекта класса *PrintMsg* с помощью оператора *new*:

```
ptr = new PrintMsg(10, 5, "Множественное ");
```

Деструкторы вызываются в обратном порядке.

### 3.2.2 Устранение неоднозначностей

Если в двух базовых классах будут объявлены одинаковые по сигнатуре методы (или поля с одинаковым именем), то, когда потребуется вызвать данный метод в классе-наследнике, возникнет вопрос: какой из унаследованных методов при этом будет использован? Ведь методы, объявленные в базовых классах, имеют одинаковые имена и сигнатуры. В результате при компилировании программы возникнет неоднозначность, которую необходимо устранить, иначе компилятор вернет сообщение об ошибке.

Неоднозначность можно устранить явным обращением к необходимой функции:

***Ob.ClassName::Func();***

В любом случае при возникновении подобной ситуации, когда необходимо сделать выбор между одноименными методами или полями различных базовых классов, следует явно указывать имя необходимого базового класса перед именем функции-члена или переменной.

### 3.2.3 Проблемы множественного наследования

Хотя множественное наследование имеет ряд преимуществ по сравнению с одиночным, программисты неохотно используют его. Основная проблема состоит в том, многие компиляторы C++ не поддерживают множественное наследование; это затрудняет отладку программы, тем более что все возможности, реализуемые этим методом, можно получить и без него.

Рекомендуется использовать множественное наследование когда новый класс нуждается в функциях и возможностях из более чем одного базового класса.

### 3.2.4 Классы возможностей

Промежуточным решением между одиночным и множественным наследованием классов является использование ***классов возможностей***. Так, класс *Horse* может происходить от двух базовых классов — *Animal* и *Displayable*, причем последний добавляет лишь несколько методов отображения объектов на экране.

Методы класса возможностей передаются в производные классы с помощью обычного наследования. Единственное отличие классов возможностей от других классов состоит в том, что они практически не содержат никаких данных. Различие довольно субъективное и отражает лишь общую тенденцию программирования, сводящуюся к тому, что добавление функциональных возможностей классам не должно сопровождаться усложнением программы.

## 3.3 Порядок выполнения лабораторной работы

3.3.1 В ходе самостоятельной подготовки изучить основы работы с “дружественными” классами и функциями.

3.3.2 Разработать программу на языке C++. Работа программы должна быть реализована в виде меню со следующими пунктами:

- создание объектов базовых классов;
- создание объекта производного класса;

- вызов уникального метода производного класса;

- вызов объектом производного класса методов базовых классов (имена методов могут совпадать).

3.3.3 Разработать тестовые примеры.

3.3.4 Выполнить отладку программы.

3.3.5 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

3.3.6 Оформить отчет по проделанной работе.

### 3.4 Варианты заданий

Описать интерфейс ввода-вывода. Описать иерархию классов, заданную по варианту. Для каждого класса описать конструкторы и деструктор (по необходимости), функции ввода и вывода значений полей. В каждом классе должны присутствовать минимум одно уникальное поле и один уникальный метод. Проиллюстрировать корректную работу механизма множественного наследования — для этого создать объекты базовых классов и заполнить их поля данными, вывести на печать. Создать объект класса-наследника, его поля заполнить значениями соответствующих полей базовых классов. Вывести на печать данные полученного объекта.

***Вариант 1***

Базовые классы: Юноша (ФИО, год рождения), Учащийся (№зачетки, курс, средний балл). Класс-наследник: Студент.

***Вариант 2***

Базовые классы: Человек (ФИО, год рождения), Должность (название, оклад). Класс-наследник: Служащий.

***Вариант 3***

Базовые классы: Устройство (название, мощность, производитель), Звук (частота, громкость). Класс-наследник: Сирена.

***Вариант 4***

Базовые классы: Корабль (название, водоизмещение, скорость), Транспорт (тоннаж). Класс-наследник: Сухогруз.

***Вариант 5***

Базовые классы: Вода (объем), Вещество (вес, название, тип). Класс-наследник: Раствор.

***Вариант 6***

Базовые классы: Книга (название, издательство), Автор (ФИО, год рождения). Класс-наследник: Произведение.

***Вариант 7***

Базовые классы: Прибор (цена деления, погрешность вычислений), Спектр (количество составляющих). Класс-наследник: Спектрометр.

***Вариант 8***

Базовые классы: Машина (марка, мощность, скорость), Транспорт (тоннаж, тип). Класс-наследник: Грузовик.

***Вариант 9***

Базовые классы: Устройство (название, мощность, производитель), Свет (яркость, цвет). Класс-наследник: Фонарик.

***Вариант 10***

Базовые классы: Ткань (название, цвет, цена), Стиль (название). Класс-наследник: Костюм.

***Вариант 11***

Базовые классы: Бумага (цвет, фактура, производитель), Дата (день, месяц, год). Класс-наследник: Календарь.

***Вариант 12***

Базовые классы: Колесо (диаметр, количество), Обувь (размер, цвет). Класс-наследник: Ролики.

***Вариант 13***

Базовые классы: Топливо (тип, вес), Окислитель (тип, объем).

Класс-наследник: Огонь.

***Вариант 14***

Базовые классы: Летательный аппарат (тип, скорость полета), Транспорт (тоннаж). Класс-наследник: Шаттл.

***Вариант 15***

Базовые классы: Птица (крылья, скорость), Лошадь (цвет, кличка). Класс-наследник: Пегас.

### 3.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 3.6 Контрольные вопросы

3.6.1 Объясните понятие “множественное наследование” и его применение.

3.6.2 Перечислите особенности работы конструкторов и деструкторов при множественном наследовании.

3.6.3 Поясните механизм возникновения неоднозначностей в программах с использованием множественного наследования, а также опишите пути решения данной проблемы. Приведите пример.

3.6.4 Перечислите достоинства и недостатки множественного наследования.

3.6.5 Расскажите о классах возможностей.

# 4 ЛАБОРАТОРНАЯ РАБОТА № 4

## ИССЛЕДОВАНИЕ МЕХАНИЗМА ДРУЖЕСТВЕННОСТИ

### 4.1 Цель работы

Приобретение практических навыков при написании объектно-ориентированных программ с использованием механизма дружественности.

### 4.2 Краткие теоретические сведения

#### 4.2.1 Дружественные функции

Обычный способ доступа к закрытым членам класса — использование открытой функции-члена. Однако C++ поддерживает и другой способ получения доступа к закрытым членам класса — с помощью дружественных функций. Дружественные функции не являются членами класса, но, тем не менее, имеют доступ к его закрытым членам. Более того, одна такая функция может иметь доступ к закрытым членам нескольких классов.

Чтобы объявить функцию дружественной некоторому классу, в определение этого класса включают ее прототип, перед которым ставится ключевое слово ***friend.***

```
#include < iostream.h>
 class AnyClass2; //Неполное объявление класса
 class AnyClass1
 {
    int d;                              // Закрытое (приватное!) поле
     public:
     AnyClass1(int _d) {d = _d;}        // Конструктор
                                        //Объявление дружественной функции
     friend bool IsFactor(AnyClassl ob1, AnyClass2 ob2);
 };
class AnyClass2
    { int n;                            // Закрытое (приватное!) поле
 public:
  AnyClass2(int _n) (n = _n;}
                                         //Объявление дружественной функции
  friend bool IsFactor(AnyClassl ob1, AnyClass2 ob2);
 };
                          //Определение дружественной функции
 bool IsFactor(AnyClassl ob1, AnyClass2 ob2)
 {
  if (!(ob1.n % ob2.d)) return true; // Работа с закрытыми полями объектов (!)
  else return false;
 }
```

```
main( )
 {
        AnyClassl ob1(12);
  AnyClass2 ob2(3);
  if (IsFactor(obl, ob2)) cout << «12 делится без остатка на 3\n»;
  else cout << «12 не делится 6ез остатка на 3\n»;
  return 0;
 }
```

Также эта программа демонстрирует важный случай применения неполного объявления класса — без применения этой конструкции в данном случае было бы невозможно объявить естественную функцию для двух классов. Неполное объявление класса *AnyClass2* дает возможность использовать его имя в объявлении дружественной функции еще до его определения.

Дружественная функция не является членом класса, в котором она объявлена. Поэтому, вызывая дружественную функцию, не нужно указывать имя объекта или указатель на объект и операцию доступа к члену класса (точку или стрелку). Доступ к закрытым членам класса дружественная функция получает только через объект класса, который, в силу этого, должен быть либо объявлен внутри функции, либо передан ей в качестве аргумента. Дружественная функция не наследуется, то есть она не является таковой для производных классов. С другой стороны, функция может быть дружественной сразу нескольким классам.

### 4.2.2 Дружественные классы

Для создания дружественного класса достаточно включить в объявление класса имя другого класса, объявляемого дружественным, перед которым ставится ключевое слово ***friend***.

Например:

```
class AnyClass
{  //..
   friend class AnotherClass;
 }
```

Класс не может объявить сам себя другом некоторого другого класса. Для того чтобы механизм дружественности сработал, он должен быть объявлен дружественным в другом классе, к которому нужно получить доступ.

```
#include < iostream.h>
 class B;                  // Неполное описание класса
 class A
 {
   friend class B;         //Класс В объявлен другом класса А…. (см. ниже)
   int x;
   void IncX(){x++;}
```

```
   public:
    A(){x = 0;}
};
class B
{
    A obA;  // …поэтому класс B получает доступ к закрытым полям класса A…
    public:
          void ShowValues ( );        // …и использует их в этой функции.
 };
void B::ShowValues ( )
{
     cout << «Сначала obA.x = « << obA.x<< «\n»;
     obA.IncX( );
     cout << «Затем obA.x = « << obA.x;
}
int main( )
{
    B obB;
    obB.ShowValues( );
    return O;
}
```

Два класса могут объявить друг друга друзьями. С практической точки зрения такая ситуация свидетельствует о плохой продуманности иерархии классов, тем не менее, язык C++ допускает и такую возможность. В этом случае объявления классов должны иметь вид:

```
class B; //Неполное объявление класса
class A
{   friend class B;  /*...*/ };
class B
{  friend class A;  /*...*/ };
```

Неполное объявление класса, которое приведено в данном фрагменте, может понадобиться, только если в классе A имеется ссылка на класс B, например, в параметре функции-члена.

### 4.2.3 Свойства дружественности

C++ предоставляет возможность обойти (или нарушить) одну из основополагающих концепций ООП — концепцию инкапсуляции данных — с помощью друзей. Но использовать ее без веских причин не стоит. Обычно такой метод доступа используется, например, когда из функции необходимо получить доступ к приватным полям и

методам сразу нескольких классов одновременно. Однако если есть возможность решить проблему без использования дружественности – то лучше так и сделать.

По отношению к дружественным классам действуют следующие правила:

- дружественность не является взаимным свойством: если класс B друг класса A, то это не означает, что класс A является другом для класса B;
- дружественность не наследуется: если класс B друг класса A, то классы, производные от B, не являются друзьями A;
- дружественность не распространяется на потомков базового класса: если класс B друг класса A, то B не является другом для классов, производных от A.

### 4.3 Порядок выполнения лабораторной работы

4.3.1 В ходе самостоятельной подготовки изучить основы работы с “дружественными” классами и функциями.

4.3.2 Разработать программу на языке C++, которая демонстрирует использование дружественной функции. Программа должна содержать описание двух классов и вызов дружественной функции.

4.3.3 Разработать тестовые примеры.

4.3.4 Выполнить отладку программы.

4.3.5 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

4.3.6 Оформить отчет по проделанной работе.

### 4.4 Варианты заданий

Описать заданные по варианту классы (содержащие private поля и методы). Для каждого класса описать конструктор по умолчанию и конструктор с параметрами, а также деструктор (по необходимости). Создать функцию, дружественную обоим классам, и в ней обратиться к их закрытым полям и методам.

**Вариант 1**

Создать два класса: Вектор (*int* *) и Матрица (*int* **). Определить функцию умножения матрицы на вектор как дружественную. Учесть проверку соответствия размерностей.

**Вариант 2**

Создать два класса: Вещественное_число (*float*) и Матрица (*float* **). Определить функцию умножения матрицы на число как дружественную.

**Вариант 3**

Создать два класса: Вектор (*int* *) и Матрица (*int* **).

Описать функцию, определяющую сумму их максимальных значений, как дружественную.

**Вариант 4**

Создать два класса: Число (*int*) и Матрица (*int* **). Описать дружественную функцию, проверяющую на равенство число и сумму элементов на главной диагонали матрицы.

**Вариант 5**

Создать два класса: Вектор (*int* *) и Матрица (*int* **). Описать дружественную функцию, проверяющую на равенство их минимальные элементы.

**Вариант 6**

Создать два класса: Символ(char) и Слово (char *). Описать дружественную функцию, определяющую, содержится ли символ в слове (если да, то сколько раз).

**Вариант 7**

Создать два класса: Число (*int*) и Матрица (*int* **). Описать дружественную функцию, проверяющую на равенство число и минимальный элемент матрицы.

**Вариант 8**

Создать два класса: Вектор (*int* *) и Матрица (*double* **). Описать дружественную функцию, заменяющую все элементы главной диагонали матрицы на соответствующие элементы вектора. Учесть проверку соответствия размерностей.

**Вариант 9**

Создать два класса: Формат (содержит параметры форматирования – точность, ширину поля вывода, знак заполнения и проч.) и Число (*float*). Описать дружественную функцию, выводящую число на экран в соответствии с форматом вывода.

**Вариант 10**

Создать два класса: Матрица_целых_чисел (*int* **)* и Матрица_вещественных_чисел (*double* **).* Описать дружественную функцию, вычисляющую поэлементную сумму матриц и выводящую результат на экран. Учесть проверку соответствия размерностей.

**Вариант 11**

Создать два класса: Матрица (*int* **) и Координаты (две пары чисел). Описать дружественную функцию, которая меняет местами два элемента, положение которых задается координатами. Предусмотреть проверку соответствия координат и размерности матрицы.

**Вариант 12**

Создать два класса: Круг (содержит поле радиус) и Квадрат. Описать дружественную функцию, определяющую фигуру с наименьшей площадью и выводящую информацию о ней на экран.

**Вариант 13**

Создать два класса: Число (*int*) и Вектор (*int* *). Определить дружественную функцию, подсчитывающую количество элементов вектора, которые превышают по значению число.

**Вариант 14**

Создать два класса: Матрица_целых_чисел (*int* **)* и Матрица_вещественных_чисел (*double* **).* Описать дружественную функцию, вычисляющую сумму элементов расположенных на главной диагонали и сравнивающую эти суммы.

**Вариант 15**

Создать два класса: Вектор (*int* *) и Матрица (*int* **). Описать функцию, определяющую сумму положительных четных элементов в них, как дружественную.

### 4.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 4.6 Контрольные вопросы

4.6.1 Объясните понятие “дружественность”. Перечислите виды “друзей” класса.

4.6.2 Расскажите о дружественных функциях. Приведите пример.

4.6.3 Расскажите о дружественных классах. Приведите пример.

4.6.4 Объясните понятие “неполное объявление класса” и его применение.

4.6.5 Назовите положительные и отрицательные стороны использования механизма дружественности.

4.6.6 Перечислите свойства данного механизма.

# 5 ЛАБОРАТОРНАЯ РАБОТА № 5

## ИССЛЕДОВАНИЕ ПЕРЕГРУЗКИ ОПЕРАТОРОВ

### 5.1 Цель работы

Исследование назначения и средств создания перегруженных операторов при написании объектно-ориентированных программ.

### 5.2 Краткие теоретические сведения

#### 5.2.1 Перегрузка операторов

C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами класса, заданного пользователем, они выполняли свои стандартные функции. Это позволяет использовать привычные операторы совместно с объектами объявленных пользователем типов так же просто, как и с переменными стандартных типов языка C++.

Перегруженные операторы реализуются как функции с помощью ключевого слова *operator.* Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением:

Таблица 5.1 — Операции, не подлежащие перегрузке

| Оператор | Название |
|---|---|
| . | Выбор члена |
| .* | Выбор члена по указателю |
| :: | Оператор расширения области видимости |
| ? : | Оператор условия |
| # | Препроцессорный символ |
| ## | Препроцессорный символ |

Функции перегруженных операторов, за исключением *new* и *delete*, должны подчиняться следующим правилам:

- операторная функция должна быть либо нестатической функцией-членом класса, либо принимать аргумент типа класса, или аргумент, который является ссылкой на тип класса;

- операторная функция унарного оператора, объявленная как функция-член, не должна иметь параметров; если же она объявлена как глобальная функция, она должна иметь один параметр;

- операторная функция бинарного оператора, объявленная как функция-член, должна иметь один параметр; если же она объявлена как глобальная функция, она должна иметь два параметра;

- операторная функция не может изменять число аргументов или приоритеты операций и их порядок выполнения по сравнению с использованием соответствующего оператора для встроенных типов данных;

- операторная функция не может иметь параметров по умолчанию;

- операторная функция не может быть переопределена для стандартных типов данных;

- операторные функции =, (), [] и -> должны быть нестатическими функциями-членами (и не могут быть глобальными функциями);

**- за исключением** оператора присваивания**, все операторные функции класса наследуются его потомками.**

### 5.2.2 Перегрузка унарных операторов

Унарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода без параметров, при этом неявным операндом является вызвавший ее объект, который передается ей через указатель *this, например:

```
class monstr
{ …
 monstr & operator ++() {++health; return *this;}
 …
};
….
monstr Vasia;
cout « (++Vasia).get_health();
```

Если функция определяется вне класса как дружественная, то она должна иметь один параметр типа класса:

```
class monstr
{…
friend monstr & operator ++( monstr &M);
…
};
…
monstr& operator ++(monstr &M) {++M.health; return M;}
```

Операции постфиксного инкремента и декремента должны иметь первый параметр типа *int*. Он используется только для того, чтобы отличить их от префиксной формы:

```
class monstr
{…
monstr operator ++(int)
{
   monstr M(*this);
   health++;
   return M;
}
…
};
```

```
        …
monstr Vasia;
cout « (Vasia++).get_health();
```

### 5.2.3 Перегрузка бинарных операторов

Бинарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода с одним параметром, при этом вызвавший ее объект передается в функцию в качестве неявного параметра с помощью указателя *this:

```
class monstr
{…
bool operator >(const monstr &M)
 { if( health > M.health) return true;
    return false;
 }…
};
```

Если функция определяется вне класса, она должна иметь два параметра, один из которых обязательно должен быть типа класса:

```
class monstr
{…
friend bool operator >(const monstr &M1, const monstr &M2);
  …
};
…
bool operator >(const monstr &M1, const monstr &M2)
{       if( Ml.get_health() > M2.get_health()) return true;
        return false;
} …
```

### 5.2.4 Перегрузка операции присваивания

Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля, память под которые выделяется динамически, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику присваивания, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент — ссылку на присваиваемый объект.

```
const monstr& operator = (const monstr &M)
{ …                                  // Процесс копирования…
        return *this;
}
```

Проще говоря, операция присваивания **всегда** переопределяется как метод класса. Кроме того, из логики работы оператора следует, что операция присвоения **не**

**наследуется** — в самом деле, у наследника могут быть свои собственные поля, с которыми операция базового класса не сможет работать корректно.

### 5.2.5 Особенности перегрузки операторов

Перегрузка операторов создавалась как механизм, позволяющий упростить написание кода и сделать его более простым и понятным при чтении. Однако чрезмерное и бездумное применение данного механизма приводит к противоположному результату. Чтобы избежать этого, необходимо учитывать следующие рекомендации:

- Если семантически нет разницы, как определять оператор, то лучше его оформить в виде функции класса. Это более целесообразно с точки зрения объектного подхода и несет смысл связывания оператора с конкретным классом, которому он принадлежит.

- Крайне не рекомендуется менять семантический смысл оператора. Делать подобное не запрещено, однако это сильно затрудняет чтение и понимание кода. Поэтому, если необходимо выполнить действие, которому нет аналога среди множества операторов языка — лучше написать отдельную функцию.

- Тип возвращаемого значения сильно зависит от сути оператора. Логические операторы должны возвращать в худшем случае int, а в лучшем bool. Для операторов присваивания необходимо возвращать ссылку на измененный элемент. Унарные операторы возвращают сам объект или ссылку на него.

Следуя подобным простым правилам, можно существенно улучшить качество кода своей программы.

## 5.3 Порядок выполнения лабораторной работы

5.3.1 В ходе самостоятельной подготовки изучить основы работы с перегруженными операторами.

5.3.2 Разработать программу на языке C++, которая должна содержать:

- описание заданного класса с переопределенными для него операторами;
- два объекта полученного класса;
- демонстрацию работы унарных операторов на одном объекте;
- демонстрацию работы бинарных операторов над созданными объектами.

Ввод и вывод данных должен осуществляться перегруженными операторами работы с потоками.

5.3.3 Разработать тестовые примеры.

5.3.4 Выполнить отладку программы.

5.3.5 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

5.3.6 Оформить отчет по проделанной работе.

## 5.4 Варианты заданий

Для заданного по варианту класса выполнить следующие действия:

- описать конструкторы и деструктор (по необходимости);
- переопределить оператор вывода в поток <<;
- переопределить оператор ввода из потока >>;

- переопределить заданные по варианту операторы;
- предусмотреть обработку ошибок.

Создать два объекта заданного по варианту класса и на их примере продемонстрировать корректную работу всех перегруженных операторов.

**Вариант *1***

Создать класс целых чисел *MyInt*. Перегрузить операторы:

1) ! как унарный метод класса, проверяющий четность числа;

2) -- (префиксный) как унарную дружественную функцию, уменьшающую значение числа на 3;

3) + как бинарный метод класса, прибавляющий к значению объекта целое число;

4) == как бинарную дружественную функцию, проверяющую равенство двух объектов.

***Вариант 2***

Создать класс координат *Coord* (содержит поля *x* и *y*). Перегрузить операторы:

1) -- (префиксный) как унарный метод класса, уменьшает обе координаты на 1;

2) ! как унарную дружественную функцию, проверяющую, совпадают ли координаты с точкой (0, 0);

3) + как бинарный метод класса, результатом вычислений которого будут координаты точки, лежащей на середине отрезка, заданного параметрами оператора;

4) < как бинарную дружественную функцию, сравнивающую расстояния от заданных точек до начала координат.

***Вариант 3***

Создать класс вектор *FVector (float *).* Перегрузить операторы:

1) ++ (префиксный) как унарный метод класса, увеличивающий элементы вектора вдвое;

2) -- (префиксный) как унарную дружественную функцию, уменьшающую значение каждого элемента вектора на 6;

3) + как бинарный метод класса, суммирующий два вектора и сохраняющий результат в первом;

4) > как бинарную дружественную функцию, определяющую, существует ли в первом векторе такой элемент, который превосходит по величине все элементы второго вектора.

***Вариант 4***

Создать класс вещественных чисел *MyReal*. Перегрузить операторы:

1) ! как унарный метод класса, проверяющий значение числа на ноль;

2) ++ как унарную дружественную функцию увеличения числа на 1;

3) / как бинарный метод класса, который возвращает результат деления двух объектов как новый объект, а также с помощью переопределенного оператора ! устраняет возможность деления на 0;

4) != как бинарную дружественную функцию, проверяющую два объекта на неравенство.

***Вариант 5***

Создать класс координат *Coord* (содержит поля *x, y* и *delta*). Перегрузить операторы:

1) ++ как унарный метод класса, увеличивающий значение координат на шаг (*delta*);

2) ! как унарную дружественную функцию, проверяющую нулевое значение координат;

3) + как бинарный метод класса, складывающий два объекта (только координаты);

4) >= как бинарную дружественную функцию, сравнивающую среднее квадратичное значение координат двух объектов.

***Вариант 6***

Создать класс строка *Stroka*. Перегрузить операторы:

1) ! как унарный метод класса, проверяющий наличие символов в строке;

2) - как унарную дружественную функцию, удаляющую последний символ из строки;

3) + как бинарный метод класса для конкатенации двух строк;

4) == как бинарную дружественную функцию, проверяющую идентичность строк.

***Вариант 7***

Создать класс треугольник *Triangle* (хранит стороны *a, b, c*). Перегрузить операторы:

1) + как унарный метод класса, вычисляющий периметр треугольника;

2) ! как унарную дружественную функцию, проверяющую возможность существования заданного треугольника (ни одна из сторон не может быть равной сумме двух других сторон или превышать ее);

3) <= как бинарный метод класса, сравнивающий длины периметров двух треугольников;

4) == как бинарную дружественную функцию, проверяющую равенство сторон двух треугольников.

***Вариант 8***

Создать класс прямоугольник *Rectangle* (хранит стороны *a* и *b*). Перегрузить операторы:

1) ! как унарный метод класса, проверяющий, является ли прямоугольник квадратом;

2) - как унарную дружественную функцию определения разности между длинами сторон;

3) == как бинарный метод класса, сравнивающий два прямоугольника на равенство площадей;

4) + как бинарную дружественную функцию нахождения общей площади фигур.

***Вариант 9***

Создать класс круг *Circle* (содержит числовое поле — радиус). Перегрузить операторы:

1) ! как унарный метод класса, проверяющий корректность введенного радиуса (больше 0);

2) -- как унарную дружественную функцию, уменьшающую радиус на 1;

3) + как бинарный метод класса, увеличивающий радиус круга на число;

4) < как бинарную дружественную функцию, сравнивающую две окружности.

***Вариант 10***

Создать класс матрица целых чисел *IMatr (int* **). Перегрузить операторы:

1) -- как унарный метод класса, меняющий знак для отрицательных элементов матрицы;

2) ++ как унарную дружественную функцию, заменяющую нулевые элементы матрицы на максимальный элемент;

3) + как бинарный метод класса, суммирующий поэлементно две матрицы;

4) >= как бинарную дружественную функцию, поэлементно сравнивающую матрицы между собой.

***Вариант 11***

Создать класс координаты *Coords* (содержит две пары чисел (*x1, y1*) и (*x2, y2*)). Перегрузить операторы:

1) ! как унарный метод класса, проверяющий на совпадение пары координат (*x1* на равенство с *x2* и аналогично *y1* и *y2* — одновременно);

2) ++ как унарную дружественную функцию, увеличивающую первую пару координат на случайное число;

3) * как бинарный метод класса, умножающий координаты на заданное число;

4) != как бинарную дружественную функцию сравнения координат двух объектов.

***Вариант 12***

Создать класс символ *MyChar* (содержит поле *char*). Перегрузить операторы:

1) ! как унарный метод класса, проверяющий, является ли символ латинской буквой;

2) -- как унарную дружественную функцию, переводящую букву из верхнего регистра в нижний (использовать оператор ! для проверки символа);

3) += как бинарный метод класса, обменивающий символы двух объектов местами;

4) == как бинарную дружественную функцию, проверяющую символы двух объектов на равенство.

***Вариант 13***

Создать класс двумерный массив *DMatr* (*double* **). Перегрузить операции:

1) ! как унарный метод класса, проверяющий наличие нулевых элементов в матрице;

2) + как унарную дружественную функцию, вычисляющую сумму элементов матрицы;

3) * как бинарный метод класса, вычисляющий сумму поэлементного произведения двух матриц;

4) == как бинарную дружественную функцию, проверяющую равенство двух объектов.

***Вариант 14***

Создать класс время *Time* (часы, минуты, секунды). Перегрузить операции:

1) -- как унарный метод класса, уменьшающий время на 5 секунд;

2) ++ как унарную дружественную функцию, увеличивающую время на 5 секунд;

3) + как бинарный метод класса, прибавляющий время из другого объекта к текущему времени;

4) != как бинарную дружественную функцию, сравнивающую два объекта.

***Вариант 15***

Создать класс вектор *IVector (int *).* Перегрузить операции:

1) ++ как унарный метод класса, возводящий элементы вектора в квадрат;

2) -- как унарную дружественную функцию, вычитающую каждый следующий элемент вектора из предыдущего (последний элемент оставить неизменным);

3) + как бинарный метод класса, вычисляющий сумму двух векторов;

4) < как бинарную дружественную функцию, сравнивающую максимальные элементы двух векторов.

### 5.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 5.6 Контрольные вопросы

5.6.1 Опишите механизм перегрузки операторов и его предназначение.

5.6.2 Перечислите операторы, которые можно перегрузить.

5.6.3 Перечислите операторы, которые нельзя перегрузить.

5.6.4 Приведите примеры всех возможных вариантов перегрузки операторов, дайте подробные пояснения.

5.6.5 Перечислите ограничения механизма перегрузки операторов.

5.6.6 Перечислите достоинства и недостатки механизма перегрузки операторов, особенностях его применения.

5.6.7 Приведите примеры перегрузки унарного оператора.

5.6.8 Приведите примеры перегрузки бинарного оператора.

# 6ЛАБОРАТОРНАЯ РАБОТА № 6

# ИССЛЕДОВАНИЕ ШАБЛОНОВ ФУНКЦИЙ

## 6.1 Цель работы

Исследование назначения и способа описания шаблонов функций, применение их при написании объектно-ориентированных программ.

## 6.2 Краткие теоретические сведения

### 6.2.1 Шаблоны и их применение

Шаблоны позволяют давать обобщенные, в смысле произвольности используемых типов данных, определения функций и классов. Поэтому их часто называют параметризованными функциями и параметризованными классами. Часто также используются термины “шаблонные функции” и “шаблонные классы”.

Шаблон функции представляет собой обобщенное определение функции, из которого компилятор автоматически создает представитель функции для заданного пользователем типа (или типов) данных. Когда компилятор создает по шаблону функции конкретную ее реализацию, то говорят, что он создал ***порожденную функцию***. Синтаксис объявления шаблона функции имеет следующий вид:

```
template < class T_1|T_1 идент_1>, . . . , <class Tn |Tn идент_n >
возвр_тип имя_функции (список параметров)   // Заголовок функции
{       /* Тело функции */
}
```

За ключевым словом *template* следуют один или несколько параметров, заключенных в угловые скобки, разделенные между собой запятыми. Каждый параметр является либо ключевым словом *class*, за которым следует имя типа, либо именем типа, за которым следует идентификатор.

Для задания параметризованных типов данных вместо ключевого слова *class* может также использоваться ключевое слово *typename*. Параметры шаблона, следующие за ключевыми словами *class* или *typename*, называют ***параметризованными типами***. Они информируют компилятор о том, что некоторый (пока что неизвестный) тип данных используется в шаблоне в качестве параметра (в момент вызова шаблона на место такого параметризованного типа станет тип, заданный программистом). Параметры шаблона, состоящие из имени типа и следующего за ним имени идентификатора, информируют компилятор о том, что параметром шаблона является константа указанного типа. Шаблонную функцию можно вызывать как обычную, никакого специального синтаксиса не требуется.

Рассмотрим пример определения и вызова шаблонных функций:

```
#include <iostream>
template <class T>          // Шаблон функции, вычисляющей квадрат
                            // введенного значения
```

```
T Sqr(T x)                  // здесь T — некий общий тип, который будет
{                           // задан при вызове функции в теле основной
   return x*x;                          // программы
}

template < class T>         // Шаблон функции обмена двух элементов
                            // в массиве по значению их индексов
T* Swap(T* t, int indl, int ind2)
{
 T tmp = t[indl];           // Собственно обмен
 t[indl] = t[ind2];
 t[ind2] = tmp;
 return t;
}
int main ( )
{  int n=10,                // Для возведения в квадрат
   i = 2, j = 5;            // Индексы в массиве
   double d = 10.21;        // Для возведения в квадрат
 char* str = "Шаблон";      // Массив букв
 cout << "Значение n = " << n << endl
 << "Его квадрат = " << Sqr (n) << endl      // Вызов шаблона для возведения
 << "Значение d = " << d << endl                    // в квадрат целого числа
 <<"Его квадрат = " << Sqr(d) << endl        // Вызов шаблона для возведения
 << "Исходная строка = " << str << endl             // в квадрат веществ. числа
                                            // Преобразование массива
 << "Преобразованная строка = " << Swap(str, i, j) << endl;
return 0;
}
```

При выполнении программа выводит на экран следующее:

```
Значение n = 10
Его квадрат = 100
Значение d = 10.21
Его квадрат = 104.2441
Исходная строка = Шаблон
Преобразованная строка = Шанлоб
```

Обратите внимание на вызовы шаблонных функций, сделанные в предыдущем примере: *Sqr(n)* и *Sqr(d)*. Каждый такой вызов приводит к построению конкретной ***порожденной функции***. В данном случае первый вызов приводит к построению компилятором функции *Sqr()* для целого типа данных, во втором — для типа *double*. В связи с этим процесс построения порожденной функции компилятором называют ***конкретизацией шаблонной функции***.

Как и для обычных функций, можно создать прототип шаблона функции в виде его предварительного объявления. Например:

```
template < class T> T* Swap(T* t, int indl, int ind2);
```

Каждый типовой параметр шаблона должен появиться в списке формальных параметров функции. Например, следующее объявление приведет к ошибке во время компиляции:

```
template < class T, class U>
T FuncName(U);
```

Шаблонная функция может быть также объявлена внешней, статической или встраиваемой (*inline*). В этом случае соответствующий спецификатор располагается вслед за списком параметризированных типов шаблона, а не перед ключевым словом *template*:

```
template extern          //Объявление внешней шаблонной функции
T* Swap(T* t, int indl, int ind2);

template static          //Объявление статической шаблонной функции
T* Swap(T* t, int indl, int ind2);

template inline          //Объявление встроенной шаблонной функции
T *Swap(T* t, int indl, int ind2);
```

### 6.2.3 Недостатки шаблонов

Шаблоны предоставляют определенные выгоды при программировании, связанные с широкой применимостью кода и легким его сопровождением. В общем, этот механизм позволяет решать те же задачи, для которых используется полиморфизм. С другой стороны, в отличие от макросов, они позволяют обеспечить безопасное использование типов данных. Однако с их использованием связаны и некоторые недостатки:

- программа содержит полный код для всех порожденных представителей шаблонного класса или функций;
- не для всех типов данных предусмотренная реализация класса или функции оптимальна.

Преодоление второго недостатка возможно с помощью специализации шаблонов.

## 6.3 Порядок выполнения лабораторной работы

6.3.1 В ходе самостоятельной подготовки изучить основы работы с шаблонами функций и классов.

6.3.2 Разработать программу на языке C++, которая обрабатывает данные разных типов (int, char, и др.). Функция обработки данных должна быть реализована как шаблон.

6.3.3 Разработать тестовые примеры.

6.3.4 Выполнить отладку программы.

6.3.5 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

6.3.6 Оформить отчет по проделанной работе.

### 6.4 Варианты заданий

Описать функцию-шаблон, заданную по варианту. Проиллюстрировать ее корректную работу на различных по типу наборах данных (не менее трех: *int*, *char* и др.).

***Вариант 1***

Написать функцию-шаблон, выполняющую транспонирование матрицы.

***Вариант 2***

Написать функцию-шаблон, меняющую местами *i*-ю строку и *j*-й столбец в матрице.

***Вариант 3***

Написать функцию-шаблон сортировки массива по возрастанию методом пузырька.

***Вариант 4***

Написать функцию-шаблон, определяющую, существуют ли в матрице повторяющиеся строки.

***Вариант 5***

Написать функцию-шаблон, определяющую элемент, который встречается в массиве максимальное число раз.

***Вариант 6***

Написать функцию-шаблон, определяющую, существуют ли в матрице повторяющиеся столбцы.

***Вариант 7***

Написать функцию-шаблон, вычисляющую максимальное значение элемента в массиве.

***Вариант 8***

Написать функцию-шаблон, переставляющую *i*-ю и *j*-ю строки в матрице.

***Вариант 9***

Написать функцию-шаблон, записывающую массив в обратном порядке.

***Вариант 10***

Написать функцию-шаблон, меняющую диагонали матрицы местами.

***Вариант 11***

Написать функцию-шаблон последовательного поиска в массиве по ключу. Функция возвращает индекс первого найденного элемента в массиве, равного ключу.

***Вариант 12***

Написать функцию-шаблон, заменяющую в матрице все элементы A на элемент B (значения A и B передаются параметрами функции-шаблона).

***Вариант 13***

Написать функцию-шаблон, циклически сдвигающую одномерный массив элементов n раз (1-й элемент становится 2-м, 2-й – 3-м …*n*-й элемент становится первым).

***Вариант 14***

Написать функцию-шаблон, меняющую в одномерном массиве соседние элементы (поменять элементы с четными индексами на элементы с нечетными индексами).

***Вариант 15***

Написать функцию-шаблон, проверяющую матрицу на симметричность.

### 6.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 6.6 Контрольные вопросы

6.6.1 Дайте определение понятию “шаблон”, приведите пример.
6.6.2 Опишите случаи в которых выгодно применение шаблонов.
6.6.3 Перечислить особенности работы шаблонов.
6.6.4Дайте определение понятию “порожденная функция”.
6.6.5 Перечислить достоинства и недостатки шаблонов
6.6.6 Дайте определение понятию “конкретизация шаблонной функции”.
6.6.7 Дайте определение понятию “параметризованные типы”.

# 7 ЛАБОРАТОРНАЯ РАБОТА № 7
# ИССЛЕДОВАНИЕ СРЕДСТВ УПРАВЛЕНИЯ ПОТОКАМИ ВВОДА-ВЫВОДА. ИССЛЕДОВАНИЕ МЕХАНИЗМА ОБРАБОТКИ ИСКЛЮЧЕНИЙ

## 7.1 Цель работы

Изучить способы реализации и особенности управления потоками ввода/вывода, исследовать способы генерации и обработки исключений.

## 7.2 Краткие теоретические сведения

### 7.2.1 Классы потокового ввода/вывода

***Поток*** — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки C++, в отличие от функций ввода/вывода в стандартном C, обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис.

Пример:

```
#include <iostream.h>
int main()
{
 int i;
 cin >> i;                          // ввод переменной
 cout << «Вы ввели » << i;   // вывод (цепочка вывода)
 return 0;
}
```

Чтение данных из потока называется *извлечением*, вывод в поток — помещением, или *включением*. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти — буфер. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе — если буфер исчерпан.

По направлению обмена потоки можно разделить на входные (данные вводятся в память), выходные (данные выводятся из памяти) и двунаправленные (допускающие как извлечение, так и включение).

По виду устройств, с которыми работает поток, можно разделить потоки на стандартные, файловые и строковые.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, файловые потоки — для обмена информацией с файлами на внешних носителях данных (например, на магнитном диске), а строковые потоки — для работы с массивами символов в оперативной памяти.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — *ios* и *streambuf.* Класс *ios* содержит общие

для ввода и вывода поля и методы, класс *streambuf* обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами.

Таблица 7.1 — Классы потокового ввода/вывода и буферизации потоков

| | |
|---|---|
| *ios* | базовый класс потоков |
| *istream* | класс входных потоков |
| *ostream* | класс выходных потоков |
| *iostream* | класс двунаправленных потоков |
| *istringstream* | класс входных строковых потоков |
| *ostringrstream* | класс выходных строковых потоков |
| *stringstream* | класс двунаправленных строковых потоков |
| *ifstream* | класс входных файловых потоков |
| *ofstream* | класс выходных файловых потоков |
| *fstream* | класс двунаправленных файловых потоков |
| *iomanip* | класс манипуляторов двунаправленных потоков |
| *bitset* | класс для работы с двоичными числами |
| *streambuf* | базовый класс буферизации потоков |
| *filebuf* | класс буферизации файловых потоков |

Основным преимуществом потоков по сравнению с функциями ввода/вывода, унаследованными из библиотеки C, является контроль типов, а также расширяемость, то есть возможность работать с типами, определенными пользователем. Для этого требуется переопределить операции потоков.

К недостаткам потоков можно отнести снижение быстродействия программы, которое, в зависимости от реализации компилятора, может быть весьма значительным.

Заголовочный файл *<iostream>* содержит, кроме описания классов для ввода/вывода, четыре предопределенных объекта потокового ввода/вывода (табл. 7.2).

Таблица 7.2 — Стандартные потоки для ввода/вывода

| Объект | Класс | Описание |
|---|---|---|
| *cin* | *istream* | Связывается с клавиатурой (стандартным буферизованным вводом) |
| *cout* | *ostream* | Связывается с экраном (стандартным буферизованным выводом) |
| *cerr* | *ostream* | Связывается с экраном (стандартным не буферизованным выводом), куда направляются сообщения об ошибках |
| *clog* | *ostream* | Связывается с экраном (стандартным буферизованным выводом), куда направляются сообщения об ошибках |

Эти объекты создаются при включении в программу заголовочного файла *<iostream>*, при этом становятся доступными связанные с ними средства ввода/вывода. Имена этих объектов можно переназначить на другие файлы или символьные буферы.

В классах *istream* и *ostream* операции извлечения из потока >> и помещения в поток << определены путем перегрузки операций сдвига.

Операции извлечения и чтения в качестве результата своего выполнения формируют ссылку на объект типа *istream* для извлечения и ссылку на *ostream* — для включения. Это позволяет формировать цепочки операций, что проиллюстрировано последним оператором приведенного ранее примера. Вывод при этом выполняется слева направо.

Как и для других перегруженных операций, для вставки и извлечения невозможно изменить приоритеты, поэтому в необходимых случаях используются скобки:

```
cout << i + j;      // Скобки не требуются — приоритет сложения больше, чем <<
cout << (i < j);    // Скобки необходимы — приоритет операции отношения
                    // меньше, чем <<:
cout << (i << j);   // Правая операция << означает сдвиг
```

### 7.2.2 Флаги и форматирующие методы

Флаги представляют собой отдельные биты, объединенные в поле *x_flags* типа *long* класса *ios*. Флаги перечислены в таблице 7.3. Форматирующие методы приведены в таблице 7.4.

Таблица 7.3 — Флаги форматирования

| Флаг | Умолчание | Описание действия при установленном бите |
|---|---|---|
| 1 | 2 | 3 |
| *skipws* | + | При извлечении пробельные символы игнорируются |
| *left* | | Выравнивание по левому краю поля |
| *right* | + | Выравнивание по правому краю поля |
| *internal* | | Знак числа выводится по левому краю, число — по правому. |
| *dec* | + | Десятичная система счисления |
| *oct* | | Восьмеричная система счисления |
| *hex* | | Шестнадцатеричная система счисления |
| *showbase* | | Выводится основание системы счисления (0x для шестнадцатеричных чисел и 0 для восьмеричных) |
| *showpoint* | | При выводе вещественных чисел печатать десятичную точку и дробную часть |
| *uppercase* | | При выводе использовать символы верхнего регистра |
| *showpos* | | Печатать знак при выводе положительных чисел |
| *scientific* | | Печатать вещественные числа в форме мантиссы с порядком |
| *fixed* | | Печатать вещественные числа в форме с фиксированной точкой |
| *unitbuf* | | Выгружать буферы всех потоков после каждого вывода |
| *stdio* | | Выгружать буферы потоков stdout и stderr после каждого вывода |

ПРИМЕЧАНИЕ:

Флаги (*left, right* и *interna*l), (*dec, oct* и *he*x), а также (*scientific* и *fixed*) взаимно исключают друг друга, то есть в каждый момент может быть установлен только один флаг из каждой группы.

Таблица 7.4 — Функции форматирования

| Функция | Описание действия |
|---|---|
| *int width(int w);* | устанавливает ширину ноля вывода в соответствии со значением параметра |
| *int width();* | возвращает значение ширины поля вывода |
| *char fill(char);* | устанавливает значение текущего символа заполнения, возвращает старое значение символа |
| *char fill();* | возвращает текущий символ заполнения |
| *int precision(int);* | устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности |
| *int precision();* | возвращает значение точности представления при выводе вещественных чисел |
| *long flags(long f);* | установить состояния всех флагов |
| *long flags();* | вернуть состояния всех флагов |
| *long setf (long setbits, long ield);* | присваивает флагам, биты которых установлены в первом параметре, значение соответствующих битов второго параметра |
| *long setf(long);* | устанавливает флаги, биты которых установлены в параметре |
| *long unsetf(long);* | сбрасывает флаги, биты которых установлены в параметре |

Пример форматирования при выводе с помощью флагов и методов:

```
#include <iostream.h>
int main()
{ long a = 1000, b = 077;
  cout.width(7);
  cout.setf(ios::hex | ios::showbase | ios::uppercase);
  cout << a;
  cout.width(7);
  cout << b << endl;
double d = 0.12, c = 1.3e-4;
  cout.setf(ios::left);
  cout << d << endl;
  cout << c;
  return 0;
```

```
}
```

В результате работы программы в первой строке будут прописными буквами выведены переменные *a* и *b* в шестнадцатеричном представлении, под каждую из них отводится по 7 позиций (функция *width* действует только на одно выводимое значение, поэтому ее вызов требуется повторить дважды). Значения переменных *c* и *d* прижаты к левому краю поля:

```
 0X3E8 0X3F
0.12
0.00013
```

### 7.2.3 Манипуляторы

Манипуляторами называются функции, которые можно включать в цепочку операций помещения и извлечения для форматирования данных. Манипуляторы делятся на простые, не требующие указания аргументов, и параметризованные. Пользоваться манипуляторами более удобно, чем методами установки флагов форматирования (табл. 7.5).

Таблица 7.5 — Манипуляторы

| Манипулятор | Назначение | Поток |
|---|---|---|
| 1 | 2 | 3 |
| *dec* | Вывод чисел в десятичной системе счисления | Вывод |
| *endl* | Вывод символа новой строки и флэширование | Вывод |
| *ends* | Вывод нуля (*NULL*) | Ввод\вывод |
| *flush* | Флэширование | Вывод |
| *hex* | Вывод чисел в шестнадцатеричной системе счисления | Вывод |
| *oct* | Вывод чисел в восьмеричной системе счисления | Вывод |
| *resetiosflags(long f)* | Сбросить флаги, определяемые *f* | Ввод\вывод |
| *setbase(int base)* | Установить основание системы счисления | Вывод |
| *setfill(char ch)* | Установить символ заполнения *ch* | Вывод |
| *setiosflags(long f)* | Установить флаги, задаваемые *f* | Ввод\вывод |
| *setprecision(int p)* | Установить точность, равную *p* | Вывод |
| *setw(int w)* | Установить ширину поля, равную *w* | Вывод |
| *ws* | Пропуск начальных пробелов | Ввод |

### 7.2.4 Методы обмена с потоками

В потоковых классах наряду с операциями извлечения >> и включения << определены методы для неформатированного чтения и записи в поток (при этом преобразования данных не выполняются).

В таблице 7.6 приведены функции чтения, определенные в классе *istream.*

Таблица 7.6 — Функции чтения

| Функция | Описание действия |
|---|---|
| *get ()* | возвращает код извлеченного из потока символа или *EOF* |
| *get (c)* | возвращает ссылку на поток, из которого выполнялось чтение, и записывает извлеченный символ в с |
| *get(buf,num,lim='\n')* | считывает *num-1* символов или пока не встретится символ *lim* и копирует их в символьную строку *buf.* Вместо символа *lim* в строку записывается признак конца строки (*'\0'*). Символ *lim* остается в потоке. Возвращает ссылку на текущий поток |
| *getline(buf, num, lim='\n')* | аналогична функции *get*, но копирует в *buf* и символ *lim* |
| *ignore(num = 1, lim = EOF)* | считывает и пропускает символы до тех пор, пока не будет прочитано *num* символов или не встретится разделитель, заданный параметром *lim.* Возвращает ссылку на текущий поток |
| *seekg(pos)* | устанавливает текущую позицию чтения в значение *pos* |
| *seekg(offs, org)* | перемещает текущую позицию чтения на *offs* байтов, считая от трех позиций, определяемых параметром *org: ios::beg* (от начала файла), *ios::cur* (от текущей позиции) или *ios::end* (от конца файла) |
| *flushO* | записывает содержимое потока вывода на физическое устройство |
| *put (c)* | выводит в поток символ с и возвращает ссылку на поток |
| *seekg(pos)* | устанавливает текущую позицию записи в значение *pos* |

В классе *ostream* определены аналогичные функции для вывода (табл. 7.7).

Таблица 7.7 — Функции вывода

| Функция | Описание действия |
|---|---|
| *seekg (offs, org)* | перемещает текущую позицию записи на *offs* байтов, считая от трех позиций, определяемых параметром *org: ios::beg* (от начала файла), *ios::cur* (от текущей позиции) или *ios::end* (от конца файла) |
| *write(buf, num)* | записывает в поток *num* символов из массива *buf* и возвращает ссылку на поток |

### 7.2.5 Файловые потоки

Чтобы открыть для вывода файл *myfile.txt* с помощью объекта *ofstream*, необходимо создать экземпляр объекта класса ofstream и передать ему имя файла в качестве параметра: *ofstream fout («myfile.txt»).*

Чтобы открыть этот файл для ввода, применяется та же методика, за исключением того, что используется объект класса *ifstream: ifstream fin («myfile.txt»).*

Обратите внимание, что *fout* и *fin* не более чем имена объектов; здесь *fout* использовался для вывода в файл подобно тому, как *cout* используется для вывода на экран; *fin* аналогичен *cin*.

Очень важным методом, используемым в файловых потоках, является функция-член *close()*. Каждый раз при открытии файла для чтения или записи (или и того и другого) создается соответствующий объект файлового потока. По завершении работы файл необходимо закрыть (например: *fout.close()*;), чтобы впоследствии не повредить его и записанные в нем данные. На практике нередко бывают непредвиденные случаи, когда не закрытые файлы теряют всю внесенную в них информацию.

После того как объекты потока будут ассоциированы с файлами, они используются наравне с другими объектами потока ввода и вывода. Например:

```
#include <fstream.h>
int main()
 {  char buffer[255];            // для ввода данных пользователем
    cout << "File name: ";
    cin.getline (buffer,255);
    ofstream fout(buffer);        // открыть для записи
    fout << "This line written directly to the file\n";
    cin.getline (buffer,255);     // получить данные от пользователя
    fout << buffer;               // и запись их в файл
    fout.close();                 // закрыть файл
    return 0;                     // уходя, гасите свет
}
```

Обычно объект класса *ofstream*, открывая файл для записи, создает новый файл, если таковой не существует, или усекает его длину до нуля, если файл с этим именем уже существует (то есть удаляет все его содержимое). Изменить стандартное поведение объекта *ofstream* можно с помощью второго аргумента конструктора, заданного явно.

Допустимыми аргументами являются:

- *ios::app* — добавляет данные в конец файла, не усекая прежнее его содержимое;
- *ios::ate* — осуществляет переход в конец файла, но запись допускает в любом месте файла;
- *ios::trunc* — задано по умолчанию; усекает существующий файл полностью;
- *ios::nocreate* — открывает существующий файл, если его нет — ошибка;
- *ios::noreplace* — открывает несуществующий файл, иначе выдаст ошибку.

### 7.2.6 Ошибки потоков

Каждый поток (*istream* или *ostream*) имеет связанное с ним состояние. Установкой и соответствующей проверкой этого состояния вылавливаются ошибки и нестандартные ситуации. Состояние потока вводится в *basic_ios*, базовом классе класса *basic_stream*, в <*ios*> (табл 7.8).

Таблица 7.8 — Функции работы с ошибками потоков

| Функция | Описание действия |
|---|---|
| *bool good() const;* | следующая операция может выполниться |
| *bool fail() const;* | следующая операция не выполнится |
| *bool eof() const;* | виден конец ввода |
| *bool bad() const;* | поток испорчен |
| *iostate rdstate() const;* | получение флагов состояния ввода/вывода |
| *void clear(iostate f=goodbit);* | сбрасываются флаги ошибок (по умолчанию - все). Обычно после возникновения ошибки нужно сбросить ошибочное состояние, чтобы дальше пользоваться потоком, но этого недостаточно – для восстановления работоспособности еще нужно вручную очистить буфер ввода/вывода. |
| *void setstate(iostate f) {clear(rdstate() \| f);}* | добавление *f* к флагам состояния |
| *operator void*() const;* | не ноль, если *!fail()* |
| *bool operator!() const {return fail();}* | не *good()* |

Состояние потока представляется набором флагов. Эти флаги определены в базовом классе *ios*:

```
enum io_state
{  goodbit  = 0x00,      //Нет ошибок
   eofbit = 0x01,        //Достигнут конец файла
   failbit = 0x02,       //Ошибка форматирования или преобразования
   badbit  = 0x04        //Серьезная ошибка, после которой
   };                                //пользоваться потоком невозможно.
```

Пример: проще всего состояние потока можно проверить как обычное логическое выражение. Обычно нужно проверить, ввел ли пользователь то, что ожидает программа:

```
double d;
cin>>d;  /* 1) 2 – преобразуется к d=2.0
            2) 2A – d=2.0 – значение сформировано, но «A» еще осталось в буфере потока и ждет приема char
            3) 3,3 вместо 3.3 – d=3.0, но запятая и вторая 3 осталась в буфере ввода и ждет ввода
            4) вводим AAA - поток испорчен, вырабатывается флаг ошибки, пользоваться d нельзя! Значение d не изменилось!!! Весь последующий ввод (cin>>) будет проигнорирован!
         */
 if( !cin )
```

```
{           /* Сюда попадем, если только поток ввода Ваш ввод
               никаким образом проинтерпретировать не может (случай 4).
               Для того, чтобы программу можно было выполнять дальше
               необходимо:
               1) сбросить ошибки
               */
  cin.clear();  // иначе весь последующий ввод будет проигнорирован
}
else {/*используем d*/}
               // 2) очистить буфер ввода — убрать лишнее (случаи 2, 3)
cin.ignore(MAX_INT,'\n');
cin>>d;        // теперь можно вводить снова
```

### 7.2.7 Обработка исключительных ситуаций

Исключение — это некоторое событие, которое является неожиданным и зачастую прерывает нормальный процесс выполнения программы.

Стандарт языка C++ предусматривает встроенный механизм обработки исключений. Кроме того, компиляторы Visual C++ 6.0 и Borland C++ Builder предусматривают еще один механизм обработки исключений, который реализуется в тесном взаимодействии этих компиляторов с операционной системой и называется структурированной обработкой исключений (SEH — от англ. Structured Exception Handling). Этот последний механизм был реализован еще в компиляторах языка C, в связи с чем часто называется C-исключениями. Хотя структурированная обработка исключений может быть использована в C++, для программ, написанных на этом языке, рекомендуется использовать более новый механизм обработки исключений C++ (C++ EH от англ. C++ Exception Handling). Для управления исключениями в C++ используются три ключевых слова: ***try, catch*** и ***throw.***

Ключевое слово try служит для обозначения блока кода, который может генерировать исключение. При этом соответствующий блок заключается в фигурные скобки и называется *защищенным* или *try-блоком*:

```
try
{
   //Защищенный блок кода
}
```

Тело всякой функции, вызываемой из ***try***-блока, также принадлежит ***try***-блоку. Предполагается, что одна или несколько функций или инструкций из защищенного блока могут выбрасывать (или генерировать) исключение. Если выброшено исключение, выполнение соответствующей функции или инструкции приостанавливается, все оставшиеся инструкции ***try***-блока игнорируются, а управление передается вовне блока try.

Ключевое слово ***catch*** следует непосредственно за try-блоком и обозначает секцию кода, в которую передается управление, если произойдет исключение. За ключевым словом ***catch*** следует описание исключения, заключенное в круглые скобки.

Описание исключения состоит из имени типа исключения и необязательной переменной:

```
catch (<имя типа>[<перменная>])
 {
    //Обработчик исключения
 }
```

Имя типа исключения идентифицирует обслуживаемый тип исключений. Блок кода, обрабатывающего исключение, заключается в фигурные скобки и называется *catch-блоком* или *обработчиком* исключения. При этом говорят, что данный ***catch***-блок перехватывает исключения описанного в нем типа. Если исключение перехвачено, переменная получает его значение. Если вам не нужен доступ к самому исключению, указывать эту переменную не обязательно. Переменная исключения может иметь любой тип данных, включая созданные пользователем типы классов.

За одним ***try***-блоком могут следовать несколько ***catch***-блоков. Оператор ***catch***, с указанным вместо типа исключения многоточием, перехватывает исключения любого типа и должен быть последним из операторов ***catch***, следующих за ***try***-блоком. Рассмотрим простой пример обработки исключения:

```
#include < exception.h>
#include < iostream.h>
void func()
{  //Функция генерирует исключение (здесь может быть ошибка, которую надо найти)
 }
 //-------------------------------------------------------------------------------
 int main() {
     try   //Пример использования механизма обработки исключений C++
    {
       func(); //Защищенный участок кода
       return 0;
     }
     catch (const char*)  //Обработчик исключения типа const char*
     {
        cout << "Было выброшено исключение  типа const char*\n";
         return 1;
     }
     catch (...)     //Обработчик всех необработанных исключений
    {
        cout << "Было выброшено исключение  другого типа\n";
        return 1;
     }
}
```

В данном примере функция ***func( )*** может генерировать исключения. В связи с этим она включена в защищенный блок. Перехват исключений осуществляется в двух ***catch***-блоках. Первый из них перехватывает исключения, имеющие тип ***const char*,*** второй — любого другого типа.

Иногда возникает необходимость использования вложенных блоков *try/catch*. Для этого в языке C++ нет никаких препятствий, однако должно соблюдаться единственное требование: за каждым try-блоком обязательно должен стоять catch-блок.

Часто вложенные *try/catch* блоки возникают неявно, в результате создания защищенного блока в функции, которая сама находится в защищенном блоке.

К вложенным ***try/catch***-блокам приходится прибегать потому, что какая-то функция может непредвиденно привести к исключению. В этом случае простейший выход — заключить весь код в функции ***main()*** в такой блок, причем для перехвата исключения использовать обработчик вида ***catch(...)***. Затем можно использовать средства, предоставляемые компилятором для определения места в программе, вызвавшего появление исключения.

### 7.2.8 Перехватывание исключений

Если в некоторой точке выбрасывается исключение, поток выполнения программы прерывается и происходит следующее:

1. Если выброшена переменная встроенного типа или объект класса по значению, создается копия выброшенной переменной (причем в последнем случае используется конструктор копирования); если выброшена переменная по ссылке, копирование не производится.

2. Осуществляется поиск ближайшего обработчика, принимающего параметр, совместимый по типу с выброшенной переменной; если обработчик найден, управление программой передается найденному обработчику.

3. Если обработчик не найден, можно вызвать ***set_terminate()*** — предоставив обработчик завершения; в противном случае программа вызывает функцию завершения.

При поиске соответствующего типа исключения компилятор пользуется следующими правилами. Обработчик считается найденным, если:

a) тип выброшенной переменной совпадает с типом, ожидаемым обработчиком, либо может быть приведен к нему (другими словами, если тип выброшенной переменной есть ***T***, соответствующими ему признаются обработчики, перехватывающие исключение типа ***T***, ***const T, T& u const T&***);

b) тип выброшенной переменной представляет собой указатель, тип которого может быть преобразован к типу указателя, перехватываемого обработчиком;

c) если выброшенная переменная представляет собой объект некоторого класса, а тип перехватываемого исключения является базовым классом, наследуемым этим объектом как ***public***.

Следует обратить внимание на то, что компилятор ищет ближайший обработчик, принимающий параметр, совместимый по типу с выброшенной переменной. Этот момент не следует забывать и внимательно следить за порядком, в котором располагаются в программе обработчики исключений для данного try-блока. Обработчик, ожидающий исключение базового класса, скрывает обработчик производного класса — поэтому обязан быть описан ниже обработчика производного класса. Точно так же обработчик для указателя типа void* скрывает обработчик для указателя любого типа.

### 7.2.9 Возможности механизма исключений

В программе можно искусственно *генерировать* исключения. Для обозначения в программе места и типа выбрасываемого исключения служит ключевое слово ***throw*.** Местоположение инструкции ***throw*** определяет точку выброса исключения. Синтаксис его использования следующий:

***throw < объект >.***

Операнд в инструкции ***throw*** не является обязательным. Инструкция ***throw*** без операнда используется для повторного выбрасывания исключения того же типа, которое обрабатывается в данный момент, следовательно, она может использоваться только в catch-блоках. Рассмотрим пример, демонстрирующий выбрасывание исключений:

```
#include < stdio.h>
bool test;
class SomeException{}; \\ Класс исключений

void Func (bool bvar) \\ Функция, создающая исключительную ситуацию
{
    if (bvar) throw SomeException(); \\ Вбрасывание исключения
}

int main()
{
    try \\ Блок защищенного кода
      {
       test = true;
       Func(true) ;
      }
catch(SomeException& e)   \\ Блок обработчика исключения
     {
          test = false;
     }
    return test ? (puts("test = true.\n"),1) : (puts("test = false.\n"),0);
}
```

При выполнении программа выводит на экран: ***test = false*.**

Также, современная реализация данного механизма предполагает наличие и использование инструкции ***finally***. Все команды, отмеченные данной инструкцией, будут выполняться всегда, даже при аварийном завершении работы программы. Здесь удобно располагать наиболее важные операции, требующие обязательного верного завершения. Например, закрытие всех файлов или сохранение важной информации.

### 7.3 Порядок выполнения лабораторной работы

7.3.1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом и исключениями.

7.3.2. Разработать согласно варианту программу на языке C++, состоящую из двух частей: первая демонстрирует умение управлять потоками ввода-вывода, вторая демонстрирует умение генерировать и перехватывать исключения.

7.3.3. Разработать тестовые примеры и выполнить отладку программы.

7.3.4. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

7.3.5. Оформить отчет по проделанной работе.

### 7.4 Варианты заданий

***Вариант 1***

1. Написать программу — возведение числа m в степень числа n. Входные данные поступают с клавиатуры. Результат выводится в файл в 15-ти позициях, точность составляет 3 знака, пробелы необходимо заменить символом “@”. Предусмотреть обработку ошибок.

2. Создать класс массив букв. Описать перехват ошибок, связанных с недопустимым индексом массива. Пример: MyClass Mass(13); cout<<Mass.GetValue(27).

***Вариант 2***

1. Написать программу нахождения всех возможных делителей введенного числа. Представить результаты в виде таблицы в файле, состоящей из следующих полей: №п/п, делитель в “научном” формате, в двоичной системе, в системе счисления с основанием 16. Делители упорядочить по возрастанию значения. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = 14(\ln(x))^2 + x^2$.

***Вариант 3***

1. Написать программу вычисления длины и площади окружности. Входные данные поступают с клавиатуры. Результат выводится в файл в 17-ти позициях, точность составляет 5 знаков, пробелы необходимо заменить знаком “&”. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = \frac{3.8\sinh(x)}{36x^3}$.

***Вариант 4***

1. Найти все положительные степени двойки, значение которых не превышает величины введенного с клавиатуры числа. Входные данные поступают с клавиатуры. Результат записывается в файл в виде таблицы: №п/п, показатель степени двойки, значение в десятичной системе, значение в двоичной системе, значение в системе счисления с основанием 8. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = 99x^2 + \sinh(x)$.

***Вариант 5***

1. Написать программу решения линейного уравнения. Результаты вычислений поместить в файл. Установить ширину поля 12 символов, точность — 4 цифры, пробелы заменить на символ “%”. Предусмотреть обработку ошибок.

2. Создать класс Вектор (float *). Описать перехват ошибок, связанных с неверным вводом значений. Пример: ввели букву вместо цифры.

***Вариант 6***

1. Написать программу вычисления наибольшего общего делителя двух целых чисел. Входные данные поступают с клавиатуры. Результат записать в файл в “научном” формате и системах счисления с основаниями 10, 16. Предусмотреть обработку ошибок.

Наибольший общий делитель рекурсивно вычисляется следующим образом:

*GCD(m, n) is:*

*if m mod n equals 0 then n;*

*else GCD(n, m mod n).*

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = \frac{1}{x^2 + 4x + 4}$.

***Вариант 7***

1. Написать программу вычисления частного и остатка от деления двух целых чисел. Результаты вычислений поместить в файл. Установить ширину поля 11 символов, заменить пробелы символом “$” с помощью функций и манипуляторов. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = ctg(x) * x^2$.

***Вариант 8***

1. Написать программу преобразования температуры в градусах Цельсия (например: *15C*) в температуру в градусах по Фаренгейту (например: *59F*), и наоборот. 0 градусов по Цельсию соответствует 32 градусам по Фаренгейту. Изменение температуры на 1 градус по Цельсию соответствует изменению на 1.8 градуса по Фаренгейту. Результаты вычислений поместить в файл. Установить ширину поля 13 символов, точность — 4 цифры, заменить пробелы символом “*/*” с помощью функций и манипуляторов. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = 10\arcsin(10x + 2.2)$.

***Вариант 9***

1. Написать программу решения квадратного уравнения. Ввод данных с консоли, вывод осуществить в файл. Вывод результата в “научном” формате. Установить

ширину поля 12 символов, установить точность 4 цифры, заменить пробелы символом “~” с помощью функций и манипуляторов. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = \frac{10x}{\arccos(x^2)}$.

***Вариант 10***

1. Написать программу, печатающую символы от “*A*” до введенного с клавиатуры символа (последний возможный “*Z*”). Для каждого символа вывести номер, сам символ, шестнадцатеричный, восьмеричный и двоичный код этого символа. Сохранить результат работы программы в файле. Предусмотреть обработку ошибок.

2. Создать класс МойФайл, содержащий строку – путь к файлу. Описать перехват ошибок, связанных с некорректной работой с файлом. Пример: попытка открыть файл, не существующий по данному пути.

***Вариант 11***

1. Написать программу вычисления длины периметра и площади прямоугольника. Входные данные поступают с клавиатуры. Результат сохраняется в файле в формате: 15 позиций; точность 5 символов; заполняющий символ “#”. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = 12x + \sqrt{x-8}$.

***Вариант 12***

1. Написать программу, которая обрабатывает все символы введенной строки. Каждый символ печатается в файл в новой строке, также печатаются его двоичный, десятичный и восьмеричный коды. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = \frac{\sqrt{x^3}}{tg(x)}$.

***Вариант 13***

1. Написать программу “калькулятор” (операции: сложение, вычитание, умножение и деление). Входные данные поступают с клавиатуры в формате “число операция число”. Результат сохраняется в файле в формате: 15 позиций; точность 5 символов; заполняющий символ “^”. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = \ln(\cos(1/x))$.

***Вариант 14***

1. Написать программу, рассчитывающую элементы ряда Фибоначчи. Каждый элемент этого ряда равен сумме двух предыдущих: $X_n = X_{(n-1)} + X_{(n-2)}$. Полагать $X_0 = 1$ и $X_1 = 1$. Вычислять до номера элемента ряда, введенного с клавиатуры. Вывести номер элемента ряда, его значение, шестнадцатеричный, восьмеричный, двоичный код. Результат работы программы сохранить в файл. Предусмотреть обработку ошибок.

2. Найти значение математического выражения, описав перехват ошибок вычислений: $y = \ln(1/\arctan(x))$.

***Вариант 15***

1. Написать программу вычисления длины периметра и площади прямоугольного треугольника. Входные данные поступают с клавиатуры. Результат сохраняется в файл в формате: 18 позиций; точность 6 символов; заполняющий символ "*". Предусмотреть обработку ошибок.

2. Перехватить исключение типа переполнение значения переменной, например, при вычислении уравнения для целочисленного типа результата: $y = 10x^{10}$.

### 7.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 7.6 Контрольные вопросы

7.6.1 Объясните понятие "поток ввода/вывода".

7.6.2 Поясните механизм буферизации потоков ввода/вывода.

7.6.3 Приведите примеры работы потоков ввода/вывода с различными типами данных.

7.6.4 Перечислите виды средств форматирования потоков.

7.6.5 Расскажите о функциях и флагах форматирования потоков ввода/вывода.

7.6.6 Расскажите о манипуляторах.

7.6.7 Опишите файловые потоки и режимы работы с файлами.

7.6.8 Дайте определение термину «исключение», опишите ситуации возникновения исключений.

7.6.9 Опишите работу механизма обработки исключений.

7.6.10 Расскажите о вложенных конструкциях ***try/catch,*** о правилах оформления ***catch***-блоков.

# 8 ЛАБОРАТОРНАЯ РАБОТА № 8

# ИССЛЕДОВАНИЕ ОСОБЕННОСТЕЙ РАБОТЫ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ КОНТЕЙНЕРОВ. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

## 8.1 Цель работы

Изучить способы реализации и особенности работы программ с использованием контейнеров стандартной библиотеки шаблонов.

## 8.2 Краткие теоретические сведения

### 8.2.1 Основные концепции STL

Стандартная библиотека шаблонов STL (***Standard Template Library***) состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов для этих классов.

***Контейнеры*** — это объекты, содержащие другие однотипные объекты, и предоставляющие интерфейс для манипуляции содержащимися в них объектами. Контейнерные классы являются шаблонными, поэтому хранимые объекты могут быть и встроенных, и пользовательских типов. Эти объекты должны допускать копирование и присваивание. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. В контейнерных классах реализованы такие типовые структуры данных, как стек, список, очередь и т.д.

***Обобщенные алгоритмы*** реализуют большое количество процедур, применяемых к контейнерам — например, поиск, сортировку, слияние и т. п. Однако они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам различных контейнерных классов, но также и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов first, last, задающая диапазон обрабатываемых элементов.

Реализация указанного механизма взаимодействия базируется на использовании так называемых итераторов. ***Итераторы*** — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, для последовательного продвижения по контейнеру, а также разыменовывать для получения или изменения значения элемента.

### 8.2.2 Контейнеры

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные.

***Последовательные контейнеры*** обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся векторы (vector), списки (list) и двусторонние очереди (deque). Есть еще специализированные контейнеры (или адаптеры контейнеров), реализованные на основе базовых — стеки (stack), очереди (queue) и очереди с приоритетами (priority_queue).

***Ассоциативные контейнеры*** обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multlmap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

```
vector<int> aVect; / / создать вектор aVect целых чисел (типа int )
list<Man> department; / / создать список department типа Man
```

### 8.2.3 Итераторы

***Итераторы*** — это обобщение указателей, которые позволяют программисту работать с различными структурами данных (контейнерами) единообразным способом.

Для всех контейнерных классов STL определен тип iterator, однако реализация его в разных классах разная. Например, в классе vect, где объекты размещаются один за другим, как в массиве, тип итератора определяется посредством typedef T* iterator. А вот в классе list тип итератора реализован как встроенный класс iterator, поддерживающий основные операции с итераторами.

К основным операциям, выполняемым с любыми итераторами, относятся:

– разыменование итератора: если p — итератор, то *p — значение объекта, на который он ссылается;

– присваивание одного итератора другому;

– сравнение итераторов на равенство и неравенство (== и !=);

– перемещение его по всем элементам контейнера с помощью префиксного (++p) или постфиксного (p++) инкремента.

Так как реализация итератора специфична для каждого класса, то при объявлении объектов типа итератор всегда указывается область видимости в форме имя_шаблона::, например:

```
vector<int> :: iterator iter1;
list<Man> :: iterator iter2;
```

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если i — некоторый итератор, то вместо привычной формы

for ( i=0; i < n; ++i ) используется следующая: for ( i = first; i!=last; ++i),

где first — значение итератора, указывающее на первый элемент в контейнере, а last — значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера. Операция сравнения “<” здесь заменена на операцию “!=”, поскольку операции “<” и “>” для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы begin() и end(), возвращающие адреса first и last соответственно.

Все типы итераторов в STL принадлежат пяти категорий: ***входные, выходные, прямые, двунаправленные*** итераторы и ***итераторы произвольного доступа***.

***Входные итераторы*** (InputIterator) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как программа может вводить данные из потока cin.

***Выходные итераторы*** (OutputIterator) используются алгоритмами для записи значений в контейнер, аналогично тому, как программа может выводить данные в поток cout.

Прямые итераторы (ForwardIterator) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем они позволяют и читать, и изменять данные в контейнере.

***Двунаправленные итераторы*** (BidirectionalIterator) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).

***Итераторы произвольного доступа*** (RandomAccessIterator) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

В то время как значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может быть гарантирован ввод тех же самых значений при вторичном обращении к потоку cin). Следствием является то, что любые алгоритмы, базирующиеся на входных или выходных итераторах, должны быть однопроходными. Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые ***адаптерами итераторов***. Адаптер, просматривающий последовательность в обратном направлении, называется reverse_iterator.

### 2.8.4 Общие свойства контейнеров

В таблице 8.1 приведены имена типов, определенные с помощью typedef в большинстве контейнерных классов.

Таблица 8.1 — Унифицированные типы, определенные в STL

| Поле | Пояснение |
|---|---|
| value_type | Тип элемента контейнера |
| size_type | (эквивалентен unsigned int)<br>Тип индексов, счетчиков элементов и т. д. |
| iterator | Итератор |
| constiterator | Константный итератор (значения элементов изменять запрещено) |
| reference | Ссылка на элемент |
| constreference | Константная ссылка на элемент (значение элемента изменять запрещено) |
| keytype | Тип ключа (для ассоциативных контейнеров) |
| key_compare | Тип критерия сравнения (для ассоциативных контейнеров) |

В таблице 8.2 представлены некоторые общие для всех контейнеров операции.

Таблица 8.2 — Операции и методы, общие для всех контейнеров

| Операция или метод | Пояснение |
|---|---|
| Операции равенства (==) и неравенства ( ! =) | Возвращают значение true или false |
| Операция присваивания (=) | Копирует один контейнер в другой |
| Clear | Удаляет все элементы |
| insert | Добавляет один элемент или диапазон элементов |
| Erase | Удаляет один элемент или диапазон элементов |
| size_type size() const | Возвращает число элементов |
| size_type max_size() const | Возвращает максимально допустимый размер контейнера |
| bool empty() const | Возвращает true, если контейнер пуст |
| iterator begin() | Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении) |
| iterator end() | Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены) |
| reverse_iterator begin() | Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении) |
| reverse iterator end() | Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены) |

### 8.2.5 Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл <algorthm>.

В таблице 8.3 приведены наиболее популярные алгоритмы STL.

Таблица 8.3 — Некоторые типичные алгоритмы STL

| Алгоритм | Назначение |
|---|---|
| accumulate | Вычисление суммы элементов в заданном диапазоне |
| copy | Копирование последовательности, начиная с первого элемента |
| count | Подсчет количества вхождений значения в последовательность |
| count_if | Подсчет количества выполнений условия в последовательности |
| equal | Попарное равенство элементов двух последовательностей |
| fill | Замена всех элементов заданным значением |
| find | Нахождение первого вхождения значения в последовательность |
| find_first_of | Нахождение первого значения из одной последовательности в другой |
| find_if | Нахождение первого соответствия условию в последовательности |
| for_each | Вызов функции для каждого элемента последовательности |
| merge | Слияние отсортированных последовательностей |
| remove | Перемещение элементов с заданным значением |
| replace | Замена элементов с заданным значением |
| search | Нахождение первого вхождения в первую последовательность второй последовательности |
| sort | Сортировка |
| swap | Обмен двух элементов |
| transform | Выполнение заданной операции над каждым элементом последовательности |

В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала [first, last), где first — итератор, указывающий на начало диапазона, а last — итератор, указывающий на выход за границы диапазона.

### 8.2.6 Использование последовательных контейнеров

К основным последовательным контейнерам относятся *вектор* (vector), *список* (list) и *двусторонняя очередь* (deque).

Чтобы использовать последовательный контейнер, нужно включить в программу соответствующий заголовочный файл:

```
#include <vector>
#include <list >
#include <deque>
```

```
using namespace std;
```

Контейнер ***вектор*** является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [ ] или метода at. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер ***список*** организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n -1 элементов.

Контейнер ***двусторонняя очередь (дек)*** во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

```
vector<int> vec1;
list<string> list1;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<string> vec1(100);
list<double> list1(20);
```

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<string> vec1 (100, "Hello!");
deque<int> dec1(300, -1);
```

4. Создать контейнер и инициализировать его элементы значениями диапазона [first, last ) элементов другого контейнера:

```
int arr[7] = {15, 2, 19, -3, 28, 6, 8};
vector<int> v1(arr, arr+7);
list<int> lst(v1.beg() + 2, v1.end());
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого однотипного контейнера:

```
 vector<int> v1;
// добавить в v2 элементы
vector<int> v2(v1);
```

Для вставки и удаления последнего элемента контейнера любого из трех рассматриваемых классов предназначены методы push_back() и pop_back(). Кроме того, список и очередь (но не вектор) поддерживают операции вставки и удаления первого элемента контейнера push_front() и pop_front(). Учтите, что методы pop_back() и pop_front() не возвращают удаленное значение. Чтобы считать первый элемент, используется метод front(), а для считывания последнего элемента — метод back(). Кроме этого, все типы контейнеров имеют более общие операции вставки и удаления, перечисленные в таблице 8.4.

Таблица 8.4 — Методы insert() и erase()

| Метод | Пояснение |
|---|---|
| insert (iterator position, const T& value) | Вставка элемента со значением value в позицию, заданную итератором position |
| insert (iterator position, size_type n, const T& value) | Вставка n элементов со значением value, начиная с позиции position |
| template <class InputIter> void insert(iterator position, InputIter first, InputIter last ) | Вставка диапазона элементов, заданного итераторами first и last, начиная с позиции position |
| erase(iterator position) | Удаление элемента, на который указывает итератор position |
| erase(iterator first, iterator last ) | Удаление диапазона элементов, заданного позициями first и last |

### 8.2.7 Использование ассоциативных контейнеров

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска. Поиск производится с помощью ключей, обычно представляющих собой одно числовое или строковое значение.

В множестве (set) хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса Man, упорядоченные в алфавитном порядке по значению ключевого поля name. Если в множестве хранятся значения одного из встроенных типов, например int, то ключом является сам элемент.

Словарь (map) можно представить себе как таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения.

И во множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу). Мультимножества (multiset) и мультисловари (multimap) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют много общих методов с последовательными контейнерами. Тем не менее, некоторые методы, а также алгоритмы характерны только для них — в основном это операции, связанные с поиском информации. Рассмотрим пример реализации контейнера map. Задача: необходимо подсчитать количество вхождений каждого слова в файл.

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>

int main()
{
   const char* filename = "file.txt";  // Создаем файловый поток
   std::ifstream in_file(filename);

   std::map<std::string, int> m;
       // Данная конструкция создает контейнер map с ключевым полем
             // в виде строки (слова) и полем-счетчиком его повторов

   std::string token;      // Для чтения и хранения слов
   while(in_file >> token)      // Чтение до конца файла
      ++m[token];               // Внесение слов в контейнер, их подсчет

       // Печать содержимого контейнера с помощью итератора
   for(std::map<std::string, int>::iterator iter = m.begin(); iter != m.end(); ++iter)
      std::cout << iter->first << " -> " << iter->second << '\n';
}
```

В данном примере особое внимание стоит уделить последним двум строчкам — печати содержимого контейнера на экран. Здесь конструкция вида `std::map<std::string, int>::iterator iter = m.begin()` по своей сути есть объявление итератора `iter` для контейнера `map` и его начальная инициализация с помощью получения начального значения `first` вызовом функции `m.begin()`. Далее, с помощью функции `m.end()` устанавливается конечное значение итератора — оно же и будет условием прекращения вычислений. Последняя строчка представляет собой простой вывод пар «слово -> количество повторов» на экран.

**8.3 Порядок выполнения лабораторной работы**

8.3.1 В ходе самостоятельной подготовки изучить основы работы с контейнерами.

8.3.2 Разработать программу на языке C++ согласно выданному варианту.

8.3.3 Разработать тестовые примеры и выполнить отладку программы.

8.3.4 Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

8.3.5 Оформить отчет по проделанной работе.

### 8.4 Варианты заданий

***Вариант 1***

Написать программу, моделирующую управление каталогом в файловой системе. Для каждого файла в каталоге содержатся следующие сведения: имя файла, дата создания, количество обращений к файлу.

Программа должна обеспечивать:

– начальное формирование каталога файлов;

– вывод каталога файлов;

– удаление файлов, дата создания которых раньше заданной;

– выборку файла с наибольшим количеством обращений.

Выбор моделируемой функции должен осуществляться с помощью меню. Для представления каталога использовать контейнерный класс list из STL.

***Вариант 2***

Написать программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута.

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

– начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);

– имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;

– имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;

– вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте.

Для представления необходимых списков использовать контейнерный класс deque.

***Вариант 3***

Написать программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета.

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

– добавление заявок в список;

– удаление заявок;

– вывод заявок по заданному номеру рейса и дате вылета;

– вывод всех заявок.

Для хранения данных использовать контейнерный класс list.

***Вариант 4***

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

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

– добавление данных о книгах, вновь поступающих в библиотеку;

– удаление данных о списываемых книгах;

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

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

Хранение данных организовать с применением контейнерного класса deque.

***Вариант 5***

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

Хранение данных организовать с применением контейнерного класса list.

***Вариант 6***

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район.

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

– ввод заявки на обмен;

– поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;

– вывод всей картотеки.

Для хранения данных картотеки использовать контейнерный класс deque.

***Вариант 7***

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления.

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

– первоначальный ввод данных в информационную систему (с клавиатуры или из файла);

– вывод сведений по всем поездам;

– вывод сведений по поезду с запрошенным номером;

– вывод сведений по тем поездам, которые следуют до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса list.

***Вариант 8***

Написать программу «Работа деканата». В базе хранится информация о студентах — ФИО, оценки, группа, курс.

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

– загрузка «базы данных» (из файла) или ее создание;

– общий вывод информации о студентах;

– вывод информации о студентах запрошенной группы, курса;

– вычисление и вывод информации о среднем балле групп.

Хранение данных организовать с применением контейнерного класса deque.

***Вариант 9***

Написать программу «Инженерный цех». В базе хранится информация о работниках — ФИО, отдел, должность, стаж, зарплата.

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

– загрузка «базы данных» (из файла) или ее создание;

– общий вывод информации о работниках;

– вывод информации о заслуженных работниках каждого отдела (по стражу);

– вычисление и вывод информации о средней зарплате в отделах.

Хранение данных организовать с применением контейнерного класса list.

***Вариант 10***

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит фамилию и инициалы заявителя, а также сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район.

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

– ввод заявки на обмен;

– поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;

– вывод всей картотеки.

Хранение данных организовать с применением контейнерного класса deque.

***Вариант 11***

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов.

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

– загрузка «базы данных» словаря (из файла);

– выбор режима работы: “англо-русский” или “русско-английский”;

– вывод вариантов перевода заданного английского слова;

– вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа multimap.

***Вариант 12***

Написать программу «Моя записная книжка». Запись содержит набор полей: дата, название события, пометки, отметка о выполнении. Дата – уникальное поле. Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку, добавления и удаления записей, отображения их на экране.

Хранение данных организовать с применением контейнерного класса set.

***Вариант 13***

Написать программу — каталогизатор книг в библиотеке. Сведения о книгах представлены в следующем виде: фамилия и инициалы автора (ключевое поле), название, расположение в библиотеке (№ отдела, № стенда и № полки).

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

– добавление данных о книгах с клавиатуры или из файла;

– удаление данных о списываемых книгах;

– поиск и выдачу информации о положении книги в библиотеке;

– выдача сведений о всех книгах указанного автора.

Хранение данных организовать с применением контейнерного класса multiset.

***Вариант 14***

Написать программу «Отдел кадров». В базе хранится информация о работниках — ФИО (ключ), номер личного дела, контактная информация и др.

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

– загрузка «базы данных» (из файла) или ее создание;

– общий вывод информации о сотрудниках;

– вывод информации о сотруднике, его личном деле;

– изменение информации о сотруднике, корректировка личного дела.

Хранение данных организовать с применением контейнерного класса map.

***Вариант 15***

Дан текстовый файл. Написать программу, которая выполняет следующее:

– открывает файл на чтение;

– определяет список различных слов в нем (использовать set);

– определяет количество вхождений каждого слова в файл (использовать map);

– выводит результаты работы на экран.

### 8.5 Содержание отчета о выполнении лабораторной работы

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

### 8.6 Контрольные вопросы

8.6.1 Опишите библиотеку STL, ее назначение и содержание.

8.6.2 Дайте определение понятия “контейнер”. Назовите виды контейнеров.

8.6.3 Опишите особенности организации и работы последовательных контейнеров. Приведите примеры таких контейнеров.

8.6.4 Опишите особенности организации и работы ассоциативных контейнеров. Приведите примеры таких контейнеров.

8.6.5 Дайте определение понятия “итератор”, перечислите их виды. Приведите примеры работы итераторов.

8.6.6 Объясните термин “обобщенные алгоритмы”. Приведите примеры таких алгоритмов.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч Г. Объектно-ориентированное проектирование с примерами применения / Г.Буч. — М.: Конкорд, 1992. — 519 с.

2. Павловская Т.А. С/ С++. Программирование на языке высокого уровня [Текст] : учебник для студентов вузов, обучающихся по направлению "Информатика и вычислительная техника" / Т. А. Павловская. - Москва ; Санкт-Петербург ; Нижний Новгород : Питер, 2006. - 460 с.

3. Павловская Т. А. С++. Объектно-ориентированное программирование [Текст] : практикум : учеб. пособие для студ. вузов, обуч. по напр. подгот. диплом. спец. "Информатика и вычисл. техника" / Т. А. Павловская, Ю. А. Щупак. - М. и др. : Питер, 2005. - 265 с. : ил. - (Учебное пособие). - Библиогр.: с. 260. - Алф. указ.: с. 261-264

4. Дейтел Х. М. Как программировать на С++ (полное издание) / Х. М. Дейтел, П. Дж. Дейтел.; под ред. В.В.Тимофеева. — М.: Бином, 2008. — 1454 с.

5. Лаптев В.В. С++. Объектно-ориентированное программирование. Задачи и упражнения. / В.В. Лаптев, А.В. Морозов, А.В. Бокова.; под ред. В.В. Лаптева. — СПб.: Питер, 2007. — 288с.

6. Иванова Г.С. Объектно-ориентированное программирование: учеб для студ. вузов/ Г.С. Иванова, Т.Н. Ничушкина, Е.К. Пугачев; под ред. Г.С.Ивановой. — М.: Изд-во МГТУ им. Н.Э.Баумана, 2001. — 320 с.

*Заказ № _____ от «_____» _____2017г. Тираж _____экз.*
*Изд-во СевГУ*