



**Министерство образования и науки Украины
Севастопольский национальный технический университет**

**Методические указания
к лабораторным работам по дисциплине
“Объектно-ориентированное программирование”**

**для студентов дневной и заочной формы обучения
направления 050101 — “Компьютерные науки”**

Часть 1

**Севастополь
2013**

УДК 004.42 (075.8)

Методические указания к лабораторным работам по дисциплине “Объектно-ориентированное программирование” для студентов дневной и заочной формы обучения направления 050101 — “Компьютерные науки”: в 2 ч. / СевНТУ ; сост. **Ю.В. Доронина, Т. И. Сметанина, А. К. Забаштанский** — Севастополь: Изд-во СевНТУ, 2013. ч. 1. — 44 с.

Цель указаний: оказать помощь студентам направления “Компьютерные науки” в выполнении лабораторных работ по дисциплине “Объектно-ориентированное программирование” на языке C++.

Методические указания составлены в соответствии с требованиями программы дисциплины “Объектно-ориентированное программирование” для студентов дневной и заочной формы обучения направления 050101 — “Компьютерные науки” и утверждены на заседании кафедры Информационных систем протоколом 13 от 15 мая 2013 г.

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент:

Кожаев Е.А., канд. техн. наук, доцент кафедры кибернетики и вычислительной техники.

СОДЕРЖАНИЕ

Введение	4
Цели и задачи лабораторных работ	4
Выбор вариантов и график выполнения лабораторных работ	4
Требования к оформлению отчета	5
Лабораторная работа № 1	6
Лабораторная работа № 2	15
Лабораторная работа № 3	26
Лабораторная работа № 4	35
Библиографический список	43

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 окончании 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 часа.

В лабораторных работах студент решает заданную индивидуальным вариантом задачу. Варианты заданий приведены в каждой лабораторной работе и уточняются с преподавателем. Номер варианта 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 нужно 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 тестовый пример;
- 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 — 6-ая неделя осеннего семестра;
- лабораторная работа №3 — 10-ая неделя осеннего семестра;
- лабораторная работа №4 — 14-ая неделя осеннего семестра;
- лабораторная работа №5 — 2-ая неделя весеннего семестра;
- лабораторная работа №6 — 6-ая неделя весеннего семестра;
- лабораторная работа №7 — 10-ая неделя весеннего семестра;
- лабораторная работа №8 — 14-ая неделя весеннего 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 тестового примера.

ЛАБОРАТОРНАЯ РАБОТА №1

КЛАССЫ. КОНСТРУКТОРЫ И ДЕКТРУКТОРЫ. КОНСТРУКТОРЫ КОПИРОВАНИЯ.

1. ЦЕЛЬ РАБОТЫ

Изучить основные средства определения класса, создания объектов класса, работы с такими объектами и их уничтожения после использования.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

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>
public:
  <описание доступных элементов>
};
```

Описание заканчивается точкой с запятой.

Спецификаторы доступа *private* и *public* управляют видимостью элементов класса. Элементы, описанные после служебного слова *private*, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса описывается после спецификатора *public*. Спецификатор *protected* говорит о том, что данные-члены и функции-члены доступны для функций-членов данного класса и классов, производных от него. 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* и *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 в 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.

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 и подчиняется общим правилам C++:

```
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 [1].

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.

2.3. Конструкторы

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

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

- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).

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

- Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

- 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.

- Конструкторы не наследуются.

- Конструкторы нельзя описывать с модификаторами *const*, *virtual* и *static*.

- 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).

- 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 нуля-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:

```

CAT(const CAT & theCat);

```

В данном случае конструктор *CAT* принимает постоянную ссылку на объект класса *CAT*, 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*). Хотя оно и подходит для большинства случаев, могут возникнуть серьезные проблемы, если в классе полями 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*.

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;
- может быть виртуальным.

Деструктор должен выглядеть так:

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

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основные средства языка C++ определения класса, создания объектов класса, работы с такими объектами и их уничтожения после использования.
2. Разработать программу на языке C++.
3. Разработать тестовые примеры.
4. Выполнить отладку программы.
5. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.
6. Оформить отчет по проделанной работе.

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 (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 ошибки.

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 расписание дня: час (0-23), минуты (0-59), задача (строка). Предусмотреть функции добавления элементов в стек и удаления из него с сохранением упорядоченности по времени.

Вариант 3

Динамическая структура — одномерный массив. Тип элементов — целочисленный. Предусмотреть функции сортировки массива по возрастанию и заполнения его случайными числами.

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 — одномерный массив. Тип данных — символьный. Предусмотреть функции заполнения массива случайными литерами и поиска наиболее часто встречаемой литеры.

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 — одномерный массив. Тип данных — вещественный. Предусмотреть функции заполнения массива случайными числами и поиска среднего значения по массиву.

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 — двумерный массив, квадратная матрица. Тип данных — целочисленный. Предусмотреть функции заполнения массива случайными числами и поиска максимального и минимального элементов массива.

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 — двумерный массив, квадратная матрица. Тип данных — вещественный. Предусмотреть функции заполнения массива случайными числами и обмена строк с минимальной и максимальной суммой элементов между собой.

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. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Объясните понятие “класс”.
2. Объясните понятия “данные-члены” и “функции-члены” класса.
3. Для чего необходима операция расширения области видимости?
4. Объясните понятие “инкапсуляция”.
5. Поясните значение директив *public*, *protected* и *private*.
6. Как осуществляется доступ к членам класса?
7. Как определяется конструктор класса и для чего он используется?
8. Как определяется конструктор копирования и для чего он нужен?
9. Как определяется деструктор класса и для чего он используется?

ЛАБОРАТОРНАЯ РАБОТА № 2

ПРОСТОЕ НАСЛЕДОВАНИЕ. ВИРТУАЛЬНЫЕ ФУНКЦИИ.

1. ЦЕЛЬ РАБОТЫ

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

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

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 *private*, *protected* или *public*. 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)
{
    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
<i>Public</i> <i>Protected</i> <i>Private</i>	<i>Public</i>	<i>Public</i> <i>Protected</i> Нет доступа
<i>Public</i> <i>Protected</i> <i>Private</i>	<i>Protected</i>	<i>Protected</i> <i>Protected</i> Нет доступа
<i>Public</i> <i>Protected</i> <i>Private</i>	<i>Private</i>	<i>Private</i> <i>Private</i> Нет доступа

Можно сделать вывод, что спецификатор наследуемого доступа устанавливает тот уровень доступа, до которого понижается уровень доступа к членам, 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 (дружественные функции).

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* ptr;
    ptr = new IncMyInt(10); // Создание объекта в куче
    ptr->ShowX( );
    delete ptr;             //Удаление объекта в куче
    return 0;
}
```

В этом случае функция-член *ShowX()* объекта *ptr* 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 >

Это дает возможность 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()
{
    int z ;
    z = MyInt::GetX() ; //Вызов переопределенной функции
    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 для построения конструктора производного класса 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 вся работа по организации соответствующего вызова возлагается на 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 *daemon*, 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).

Виртуальные функции — это функции, которые гарантируют, что будет вызвана правильная функция для объекта безотносительно к тому, 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.

Чисто виртуальный метод содержит признак “= 0” вместо тела, например:

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

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять 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 **абстрактным**. 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.

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом [4].
2. Разработать программу на языке C++.
3. Разработать тестовые примеры и выполнить отладку программы.
4. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.
5. Оформить отчет по проделанной работе.

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 $a_j = a_0 + jd, j = 0, 1, 2, \dots$

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

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

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 Фигура, и производные классы: круг, прямоугольник, четырехугольник. Определить виртуальную функцию вычисления периметра.

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).

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. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое простое наследование, для чего оно применяется?
2. Что такое спецификатор доступа, для чего он нужен при наследовании?
3. Привести пример переопределения функции и объяснить суть механизма.
4. Перечислить особенности описания конструкторов и деструкторов при наследовании.
5. Что такое полиморфизм?
6. Что такое виртуальная функция? Когда она применяется? Привести пример.
7. Что такое чисто виртуальная функция? Абстрактный класс? Для чего они нужны?
8. Что такое механизм позднего связывания?

ЛАБОРАТОРНАЯ РАБОТА № 3 МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ. ДРУЖЕСТВЕННЫЕ КЛАССЫ И ФУНКЦИИ.

1. ЦЕЛЬ РАБОТЫ

Приобретение практических навыков при написании объектно-ориентированных программ. Освоение особенностей отладки объектно-ориентированных программ. Изучение механизма множественного наследования. Приобретение навыков работы с механизмом “дружественности”.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

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.

2.2. Устранение неоднозначностей

Если в двух базовых классах будет объявлены одинаковые по сигнатуре методы, 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 переменными-членами различных базовых классов, следует явно указывать имя необходимого базового класса перед именем функции-члена или переменной.

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.

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.

2.5 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;
    public:
    AnyClass1(int _d) {d = _d;}    // Конструктор
                                //Объявление дружественной функции
    friend bool IsFactor(AnyClass1 ob1, AnyClass2 ob2);
};

class AnyClass2
{ int n;
  public:
  AnyClass2(int _n) (n = _n;}
                                //Объявление дружественной функции
  friend bool IsFactor(AnyClass1 ob1, AnyClass2 ob2);
};

                                //Определение дружественной функции
bool IsFactor(AnyClass1 ob1, AnyClass2 ob2)
{
    if (!(ob1.n % ob2.d)) return true;
    else return false;
}

main( )
{
    AnyClass1 ob1(12);
    AnyClass2 ob2(3);

    if (IsFactor(ob1, ob2)) cout << «12 делится без остатка на 3\n»;
    else cout << «12 не делится без остатка на 3\n»;
    return 0;
}
```

Кроме того, эта программа демонстрирует важный случай применения неполного объявления класса: без применения этой конструкции в данном случае было бы невозможно объявить дружественную 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.

2.6. Дружественные классы

C++ предоставляет возможность обойти (или нарушить) одну из основополагающих концепций ООП — концепцию инкапсуляции данных — с помощью друзей. Однако использовать ее без веских причин не стоит. C++ позволяет объявлять два вида друзей класса: дружественную функцию или дружественный класс, предоставив им полный доступ к членам своего класса. 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 B объявлен другом класса A
    int x;
    void IncX(){x++;}
public:
    A(){x = 0;}
}
```

```

};
class B
{
    A obA;
    public:
    void ShowValues ( );
};
void B::ShowValues ( )
{
    cout << «Сначала obA.x = « << obA.x<< «\n»;
    obA.IncX( );
    cout << «Затем obA.x = « << obA.x;
}
main( )
{
    B obB;          //Конструктор по умолчанию, генерируется компилятором
    obB.ShowValues( );
    return 0;
}

```

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 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 А имеется ссылка на класс В. например, в параметре функции-члена. По отношению к дружественным классам действуют следующие правила:

- а) дружественность не является взаимным свойством: если А друг В, это не означает, что В — друг А;
- б) дружественность не наследуется: если В — друг А, классы, производные от В не являются друзьями А;
- с) дружественность не распространяется на потомков базового класса: если В — друг А, то В не является другом для классов, производных от А.

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с “дружественными” классами и функциями.
2. Разработать программу на языке C++.
3. Разработать тестовые примеры.
4. Выполнить отладку программы.
5. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.
6. Оформить отчет по проделанной работе.

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. Класс-наследник должен реализовывать описанный ранее интерфейс. Описать дружественную функцию для класса-наследника и заданного по варианту класса.

Вариант 1

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

Дружественность: класс Ведомость.

Вариант 2

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

Дружественность: класс Завод.

Вариант 3

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

Дружественность: класс Магазин.

Вариант 4

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

Дружественность: класс Расписание.

Вариант 5

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

Дружественность: класс Сосуд.

Вариант 6

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

Вариант 7

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

Вариант 8

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

Вариант 9

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

Вариант 10

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

Вариант 11

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

Вариант 12

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

Вариант 13

Базовые классы: Топливо (тип, вес), Окислитель (тип, объем).
Класс-наследник: Огонь. Дружественность: класс Жидкость.

Вариант 14

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

Вариант 15

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

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. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое множественное наследование? Привести пример.
2. Пояснить особенности работы конструкторов и деструкторов при множественном наследовании.
3. Перечислите достоинства и недостатки множественного наследования.
4. Что такое дружелюбность с точки зрения класса?
5. Какие виды друзей класса бывают?
6. Что такое дружественная функция? Привести пример.
7. Что такое дружественный класс? Привести пример.
8. Что такое неполное объявление класса и для чего это нужно?
9. Перечислите ограничения механизма дружелюбности.
10. Назовите положительные и отрицательные стороны использования механизма дружелюбности.

ЛАБОРАТОРНАЯ РАБОТА №4 ПЕРЕГРУЗКА ОПЕРАТОРОВ.

1. ЦЕЛЬ РАБОТЫ

Приобретение практических навыков при написании объектно-ориентированных программ с использованием перегруженных операторов. Освоение особенностей отладки объектно-ориентированных программ.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

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 дает возможность использовать собственные типы данных при работе с операторами точно так же, как и стандартные типы.

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 9 — Операции, не подлежащие перегрузке

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

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

- операторная функция не может быть переопределена для стандартных типов данных;
- операторная функция должна быть либо нестатической функцией-членом класса, либо принимать аргумент типа класса, или аргумент, который является ссылкой на тип класса;
- 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 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 иметь параметров по умолчанию;
- первый параметр (если он есть) операторной функции, объявленной как функция-член, должен иметь тип класса объекта, для которого вызывается соответствующий оператор: никакие преобразования типа для первого аргумента не выполняются;
- за исключением **оператора присваивания**, все операторные функции класса наследуются его потомками;
- операторные функции `=`, `()`, `[]` и `->` должны быть нестатическими функциями-членами (и не могут быть глобальными функциями).

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();

```

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:

```

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

```

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;
}

```

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить механизм перегрузки в языке C++.
2. Разработать программу на языке C++.
3. Разработать тестовые примеры.
4. Выполнить отладку программы.
5. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.
6. Оформить отчет по проделанной работе.

4. ВАРИАНТЫ ЗАДАНИЙ

Для заданного по варианту класса выполнить следующие действия:

- описать все типы конструкторов 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 стек *Stack* (содержит числовое поле). Перегрузить операторы:

- 1) -- (префиксный) как унарный метод класса, удаляющий элемент из стека;
- 2) ! как унарную дружественную функцию, проверяющую стек на наличие элементов;
- 3) + как бинарный метод класса, складывающий два объекта (поэлементно);
- 4) < как бинарную дружественную функцию, сравнивающую стеки по длине.

Вариант 3

Создать класс вектор *Vector (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 *Real*. 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 равенство.

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 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 очередь *Turn* (содержит числовое поле). Перегрузить операторы:

1) ! как унарный метод класса, проверяющий присутствие элементов в очереди;

2) -- как унарную дружественную функцию, удаляющую элемент с головы очереди;

3) + как бинарный метод класса, добавляющий элемент в конец очереди;

4) < как бинарную дружественную функцию, сравнивающую первые элементы очередей.

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 *My_Coord* (содержит две пары чисел (*x1*, *y1*) и (*x2*, *y2*)). Перегрузить операторы:

1) ! как унарный метод класса, проверяющий на совпадение пары координат (пары = координатам второй пары (*x1*, *y1*) и (*x2*, *y2*));

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

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

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

Вариант 12

Создать класс символ *My_Char* (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 местами;
- 4) `==` как бинарную дружественную функцию, проверяющую символы на равенство.

Вариант 13

Создать класс двумерный массив *Matr (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 *Vector (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. 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. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое перегрузка операторов? Для чего она предназначена?
2. Какие операторы перегрузить можно? Какие нельзя?
3. Какими способами можно перегрузить оператор? Привести пример.
4. Каким правилам подчиняются перегруженные операторы?
5. Приведите примеры перегрузки унарного оператора.
6. Приведите примеры перегрузки бинарного оператора.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч Г. Объектно-ориентированное проектирование с примерами применения / Г.Буч. — М.: Конкорд, 1992. — 519 с.
2. 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.
3. Айра Пол. Объектно-ориентированное программирование на С++ /Айра Пол. — СПб: “Невский диалект”, 2001. — 464 с.
4. Павловская Т.А. Программирование на языке высокого уровня С/С++ / Т.А. Павловская. — Питер, 2003. — 461с.
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 « _____ » _____ 2013г. Тираж _____ экз.

Изд-во СевНТУ