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
2012

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 3 ч. / СевНТУ ; сост. **Ю.В. Доронина**, **Т. И. Сметанина,** А**. К. Забаштанский, А.В. Исаева**. — Севастополь: Изд-во СевНТУ, 2012. ч. 1. — 40 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 № 7 от 23 февраля 2011 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

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 2 академических часов.

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 весеннего семестра;
- лабораторная работа №2 — 4-ая неделя весеннего семестра;
- лабораторная работа №3 — 6-ая неделя весеннего семестра;
- лабораторная работа №4 — 8-ая неделя весеннего семестра;
- лабораторная работа №5 — 10-ая неделя весеннего семестра;
- лабораторная работа №6 — 11-ая неделя весеннего семестра;
- лабораторная работа №7 — 12-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 указано 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
ИСПОЛЬЗОВАНИЕ ПОТОКОВ ВВОДА — ВЫВОДА. ПОТОКИ (IOSTREAM). ФАЙЛЫ — ПОТОКИ (FSTREAM).

1. ЦЕЛЬ РАБОТЫ

Изучить механизм потокового ввода/вывода, обеспечивающий гибкий и эффективный с гарантией типа метод обработки символьного ввода целых чисел, чисел с плавающей точкой и символьных строк, а также простую модель его расширения для обработки типов, определяемых пользователем.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1. Общие положения

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
 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, их можно разделить 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 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 (таблица 2).

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 чтения. Это позволяет формировать цепочки операций, что проиллюстрировано последним оператором приведенного примера. Вывод при этом выполняется слева направо.

Таблица 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

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
```

2.2. Флаги и форматирующие методы

Флаги представляют собой отдельные биты, объединенные в поле *x_flags* типа *long* класса *ios*. Флаги перечислены в таблице 3.

Таблица 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

Продолжение таблицы 3

1	2	3
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 4 — Функции форматирования

Функция	Описание действия
int width(int w);	устанавливает ширину поля 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
```

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.

Таблица 5 — Манипуляторы

Манипулятор	Назначение	Ввод\вывод
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

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). Ниже приведены функции чтения, определенные в классе *istream:*

Таблица 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. 1im='\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 одной из 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:

Таблица 7 — Функции вывода

Функция	Описание действия
seekg (offs, org)	перемещает текущую позицию записи на *offs* байтов, считая от одной из 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

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 пользователя
    cout << “File name: “;
    cin.getline (buffer,255);    // считать имя файла
    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;
}
```

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;
- *ios::noreplace* — открывает несуществующий файл.

2.6. Ошибки потоков

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

Таблица 8 — Функции манипуляции состоянием потока

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
    eofbi t= 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 никаким образом не
                    может проинтерпретировать ввод (см. 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
cin.ignore(MAX_INT,'\n');
cin>>d;         // теперь можно вводить снова
```

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом.

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

Вариант 1

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

2. Написать программу копирования файла: первая половина — собственно файл, вторая половина — он же в обратном порядке. Чтение происходит блоками. Обработать ошибки.

Вариант 2

1. Написать программу нахождения всех возможных делителей введенного числа. Представить результаты в виде таблицы, 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. Написать программу копирования файла с удалением лишних пробелов. Обработать ошибки.

Вариант 3

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

2. Написать программу копирования файла с удвоением пробелов. Чтение происходит блоками. Обработать ошибки.

Вариант 4

1. Найти все положительные степени двойки, значение которых не превышает величины введенного с клавиатуры числа. Входные данные поступают с клавиатуры. Результат выводится на экран 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. Написать программу копирования файла с заменой пробелов на символ “|”. Чтение происходит блоками. Обработать ошибки.

Вариант 5

1. Написать программу решения линейного уравнения. При выводе результата на экран установить ширину поля 12 символов, точность — 4 цифры, пробелы заменить на символ “%”. Предусмотреть обработку ошибок.

2. Написать программу копирования файла. В конец каждой строки файла дописать количество пробелов в строке. Чтение происходит блоками. Обработать ошибки.

Вариант 6

1. Написать программу вычисления наибольшего общего делителя двух целых чисел. Входные данные поступают с клавиатуры. Результат выводится на экран 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. Написать программу копирования файла. В выходной файл записываются только те строки, в которых есть двузначные цифры. Чтение происходит блоками. Обработать ошибки.

Вариант 7

1. Написать программу вычисления частного и остатка от деления двух целых чисел. При выводе результата на экран установить ширину поля 11 символов, заменить

пробелы символом “$” с помощью функций и манипуляторов. Предусмотреть обработку ошибок.

2. Написать программу копирования файла. В выходном файле сначала должны быть предложения, начинающиеся с гласной буквы, а потом все остальные. Чтение происходит блоками. Обработать ошибки.

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. При выводе результата на экран установить ширину поля 13 символов, точность — 4 цифры, заменить пробелы символом “/” с помощью функций и манипуляторов. Предусмотреть обработку ошибок.

2. Написать программу копирования файла. В выходном файле сначала должны быть предложения, состоящие из одного слова, затем те, что состоят из двух слов, а потом все остальные. Чтение происходит блоками. Обработать ошибки.

Вариант 9

1. Написать программу решения квадратного уравнения. Ввод и вывод — через потоки ввода-вывода. 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. Написать программу копирования файла. В выходном файле сначала должны быть предложения, заканчивающиеся вопросительным знаком, затем заканчивающиеся восклицательным знаком, а затем все остальные. Чтение происходит блоками. Обработать ошибки.

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. Предусмотреть обработку ошибок.

2. Написать программу копирования файла. В выходном файле попарно поменять слова местами. Чтение происходит блоками. Обработать ошибки.

Вариант 11

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

2. Написать программу копирования файла. В выходном файле содержатся только те предложения, в которых было найдено слово, введенное с клавиатуры. Чтение происходит блоками. Обработать ошибки.

Вариант 12

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

2. Написать программу копирования файла. В выходном файле в конец предложения дописывается количество слов в нем. Чтение происходит блоками. Обработать ошибки.

Вариант 13

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

2. Написать программу копирования файла. В выходном файле содержатся предложения, состоящие из заданного количества слов. Чтение происходит блоками. Обработать ошибки.

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. Предусмотреть обработку ошибок.

2. Написать программу копирования файла. В выходной файл записываются предложения, не содержащие запятых. Чтение происходит блоками. Обработать ошибки.

Вариант 15

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

2. Написать программу копирования файла. Входной файл содержит программу на языке Си с комментариями. В выходном файле содержится только программа, комментарии необходимо исключить. Чтение происходит блоками. Обработать ошибки.

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 “поток ввода/вывода”.
2. Поясните механизм буферизации потоков ввода/вывода.
3. Каким образом осуществляется ввод в поток?
4. Какие встроенные типы данных поддерживает оператор вставки в поток?
5. Каким образом осуществляется вывод из потока?
6. Какие средства для управления форматированием потока предусматривает библиотека ввода/вывода?
7. Какие функции-члены класса *ios* используются для опроса и установки состояния потока?
8. Какие существуют режимы открытия файла?
9. Какой режим открытия файла установлен по умолчанию?
10. Какие формы конструкторов предусмотрены для создания файлового потока?

ЛАБОРАТОРНАЯ РАБОТА №2
КЛАССЫ. КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ. 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 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.

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.

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 u 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                   // Описание класса
{
 int health, ammo;             // Поля
 color skin;
 char *name;
 public:                       // Методы
```

```
  monstr(int he = 100, int am =10);
 monstr(color sk);
 monstr(char * nam);
 int get_health(){return health;}
 int get_ammo(){return ammo;}
};
// ------------------------ описания конструкторов с параметрами
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»); // Объявление объектов
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 обой-

тись 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*). Хоть 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 распорядок 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). Предусмотреть 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 №3
ДРУЖЕСТВЕННЫЕ КЛАССЫ И ФУНКЦИИ.

1. ЦЕЛЬ РАБОТЫ

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

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

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

```
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 (!(obl.n % ob2.d)) return true;
 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;
}
```

Кроме того, 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.

2.2. 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 В объявлен другом класса А
    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 O;
}
```

Два класса могут объявить друг друга друзьями. С практической точки зрения такая ситуация свидетельствует о плохой продуманности иерархии классов, тем не менее, язык С++ 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:

a) дружественность не является взаимным свойством: если А друг В, это не означает, что В — друг А;

b) дружественность не наследуется: если В — друг А, классы, производные от В не являются друзьями А;

c) 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 С++.
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 каждого класса, заданного по варианту, необходимо описать конструктор по умолчанию и конструктор с параметрами, деструктор, функцию вывода данных, а также необходимые функции вычислений и получения значений. Предусмотреть обработку ошибок.

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 (*MyFloat*) и матрица (*float* **). Определить функцию умножения матрицы на число как дружественную.

Вариант 3

Создать два класса: вектор (*int **) и матрица (*int ***). Описать функцию, определяющую значение максимального элемента в них, как дружественную.

Вариант 4

Создать два класса: целое число (*MyInt*) и вещественное число (*Real*). Описать функцию, определяющую среднее квадратичное данных чисел, как дружественную.

Вариант 5

Создать два класса: вектор (*int **) и матрица (*int ***). Определить функцию умножения вектора на матрицу как дружественную. Учесть проверку соответствия размерностей.

Вариант 6

Создать два класса: символ (*MSymbol*) и слово (*MWord*). Описать функцию, определяющую, содержится ли символ в слове (если да, то сколько раз), как дружественную.

Вариант 7

Создать два класса: линия (*Line*, содержит параметры *a* и *b*) и круг (*Circle*, содержит координаты центра и радиус). Описать функцию, определяющую координаты точек пересечения, как дружественную.

Вариант 8

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

Вариант 9

Создать два класса: формат (*Format*, содержит параметры форматирования – точность, ширину поля вывода, знак заполнения и проч.) и число (*MyFloat*). Описать функцию, выводящую число на экран в соответствии с форматом вывода, заданным вторым классом, как дружественную.

Вариант 10

Создать два класса: матрица целых чисел (*IMatr: int ***) и матрица вещественных чисел (*DMatr: double ***). Описать функцию, вычисляющую поэлементную сумму матриц и выводящую результат на экран, как дружественную. Учесть проверку соответствия размерностей.

Вариант 11

Создать два класса: матрица (*int ***) и координаты (две пары чисел). Описать функцию, которая меняет местами два элемента, положение которых задается координатами во втором классе, как дружественную. Предусмотреть проверку соответствия координат и размерности матрицы.

Вариант 12

Создать два класса: круг (*Circle*, содержит поле радиус) и квадрат. Описать функцию, определяющую фигуру с наименьшей площадью и выводящую информацию о ней на экран, как дружественную.

Вариант 13

Создать два класса: число (*MyInt*) и вектор (*int **). Определить функцию, считающую количество элементов вектора, которые превышают по значению число, как дружественную.

Вариант 14

Создать два класса: синус (*MSin*) числа и косинус (*MCos*) числа. Описать функцию, определяющую тангенс числа, как дружественную. Предусмотреть ошибочные ситуации.

Вариант 15

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

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 дружественность с точки зрения класса?
2. Какие виды друзей класса бывают?
3. Что такое дружественная функция? Привести пример.
4. Что такое дружественный класс? Привести пример.
5. Что такое неполное объявление класса и для чего это нужно?
6. Перечислите ограничения механизма дружественности.
7. Назовите положительные и отрицательные стороны использования механизма дружественности.

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( Ml.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 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 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 C++ (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 C++ /Айра Пол. — СПб: “Невский диалект”, 2001. — 464 с.

4. Павловская Т.А. Программирование на языке высокого уровня C/C++ / Т.А. Павловская. — Питер, 2003. — 461с.

5. Лаптев В.В. C++. 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 «_____» _____2012г. Тираж _____экз.

Изд-во СевНТУ