МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к практическим занятиям
по дисциплине
«Объектно-ориентированное программирование»

для направления подготовки
09.03.02 Информационные системы и технологии
Направленность (профиль)
«Информационные системы и технологии в бизнесе»

Часть 1

Невинномысск, 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Работа на практических занятиях по учебной дисциплине «Объектно-ориентированное программирование» предполагает изучение студентами основ объектно-ориентированного программирования.

Изучение дисциплины предполагает формирование компетенции

ПК-4	Способен разработать архитектуру ИС

Выполнение практических заданий включает:

1. Изучение студентами необходимого теоретического материала по теме лабораторной работы.
2. Постановку задачи в соответствии с темой лабораторной работы и согласование ее с руководителем.
3. Построение алгоритма решения задачи и его документирование в разделе «Краткие теоретические сведения» отчета.
4. Выполнение задания.
5. Подготовку отчета о выполненной работе и его защиту.

Структура отчета по проделанной работе:

1. Тема.
2. Цель работы.
3. Постановка задачи.
4. Ход выполнения работы.
5. Блок-схема или псевдокод алгоритма решения задачи.
6. Текст программы.
7. Распечатка результатов.
8. Выводы.

Практическое занятие 1. Определение класса

Простейшее определение класса без наследования имеет вид:

```
class имя_класса  {
// по умолчанию раздел private – частные члены класса
     public:  //открытые функции и переменные класса
    };
```

Определение класса соответствует введению нового типа данных, а понятие переменной данного типа – понятию объекта (экземпляра) класса. Список членов класса включает определение данных и функций, те из них, чьи объявления находятся в описании класса, называются функциями-членами класса. В ООП для таких функций используется термин “методы класса”. Классы могут также содержать определения функций, тело которых помещается в определение класса (inline-функции). Для инициализации/уничтожения объектов используются специальные функции-конструкторы с тем же именем, что и класс, и деструкторы, с именем класса, перед которым стоит символ “~”.

Переменные, объявленные в разделе класса по умолчанию как частные (private), имеют область видимости в пределах класса. Их можно сделать видимыми вне класса, если поставить перед началом объявления слово public.

Обычно переменные в классе объявляются private-переменными, а функции видимыми (public). Открытые функции-члены класса имеют доступ ко всем закрытым данным класса, через них возможен доступ к этим данным.

Классами в C++ являются также структуры (struct) и объединения (union). Отличием структуры и объединения от класса является то, что их члены по умолчанию открытые (public), а не закрытые, как у класса. Это обеспечивает преемственность с языком C. Кроме того, структуры и объединения не могут наследоваться и наследовать.

В следующей программе определяется простейший класс Strtype, членами которого являются массив str типа char и функции set(), show(), get().

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class Strtype{
     char str[80]; //private
     public:
          void set (char *);    //задать str
        void show();            //вывести str
        char* get();            //вернуть str
            };                  //конец     определения
класса
void Strtype::set(char *s)// определение метода set()
```

```
    {
         strcpy(str,s);        //копирование s в str
    }
    void Strtype::show()           //  определение  метода
show()
    {
         cout<<str<<endl;
    }
    char * Strtype::get()      // определение метода get()
    {
         return str;
    }
    int main()
    {
         Strtype obstr;            //объявление объекта
         obstr.set("String example"); //вызов      метода
     set()
         obstr.show();                //вызов      метода
show()
         cout<<obstr.get()<<endl;
         while(!kbhit());//задержка   выхода   до   нажатия
клавиши
         return 0;
    }
```

Вывод:
```
String example
String example
```

Массив член-класса str – является частным (private), доступ к нему возможен только через функции-члены класса. Такое объединение в классе сокрытых данных и открытых функций и есть инкапсуляция. Здесь obstr – объект данного класса. Вызов функции осуществляется из объекта добавлением к имени объекта имени функции, через точку или –>, если используется указатель на объект.

Приведем примеры нескольких объявлений объектов:

```
Strtype a; //объект a
Strtype x[100]; //массив объектов
Strtype *p; //указатель на объект
p=new Strtype; // создание динамического объекта
a.set("строка"); // вызов функций
x[i].set("строка");
p->set("строка");
```

Оператор расширения области видимости “::” указывает доступ к элементам класса. Например, Strtype::set(char *s) означает принадлежность функции set(char *s) области видимости класса Strtype. Кроме этого оператор “::” используется для доступа к данным класса (оператор вида Strtype::count),

для указания внешней или глобальной области видимости переменной, скрытой локальным контекстом (оператор вида ::globalName).

Следующая программа демонстрирует создание класса Stack на основе динамического массива. Для инициализации объекта класса используется метод Stack() – конструктор класса.

```
#include <iostream.h>
#include <conio.h>
#define SIZE 10
// объявление класса Stack для символов
class Stack{
     char *stck;      // содержит стек
     int tos;         // индекс вершины стека
public:
     Stack();             //конструктор
     ~Stack(){delete [] stck;}//деструктор
     void push(char ch); // помещает в стек символ
     char pop(); // выталкивает из стека символ
};
// инициализация стека
Stack::Stack()
{
     stck=new char[SIZE];//динамический массив
     tos=0;
     cout << "работа конструктора …  \n";
}
// помещение символа в стек
void Stack::push(char ch)
{
     if (tos==SIZE)
     {
          cout << "стек полон";
          return;
     }
     stck[tos]=ch;
     tos++;
}
// выталкивание символа из стека
char Stack::pop()
{
     if (tos==0)
     {
          cout << "стек пуст";
          return 0;
     }
     tos--;
```

```
        return stck[tos];
    }
    int main()
    {
    /*образование двух автоматически инициализируемых
стеков */
        Stack s1, s2; //вызов конструктора для s1 и s2
        int i;
        s1.push('a');
        s2.push('x');
        s1.push('b');
        s2.push('y');
        s1.push('c');
        s2.push('z');
        for(i=0;i<3;i++)
            cout<<"символ из s1:"<<s1.pop() << "\n";
        for(i=0;i<3;i++)
            cout<<"символ из s2:"<<s2.pop() << "\n";
        cout.flush();
        while(!kbhit());//задержка
        return 0;
    }
```

Вывод:

```
 Работа конструктора …
 Работа конструктора …
 Символ из s1:c
 Символ из s1:b
 Символ из s1:a
 Символ из s2:z
 Символ из s1:y
 Символ из s1:x
```

Конструктор Stack() вызывается автоматически при создании объектов класса s1,s2 и выполняет инициализацию объектов, состоящую из выделения памяти для динамического массива и установки указателя на вершину стека в нуль. Конструктор может иметь аргументы, но не имеет возвращаемого значения и может быть перегружаемым.

Деструктор ~Stack() выполняет действия необходимые для корректного завершения работы с объектом, а именно, в деструкторе может освобождаться динамически выделенная память, закрываться соединения с файлами, и др. Он не имеет аргументов. Именем деструктора является имя класса, перед которым стоит знак “~” – тильда.

Рассмотрим еще один пример класса, реализующего динамический односвязный список:

```
#include <iostream.h>
#include <conio.h>
struct Node           // объявление класса Node
{
     int info;        //информационное поле
     Node* next;      //   указатель   на   следующий
элемент
};
class List                 // объявление класса List
{
     Node* top;       // указатель на начало списка
public:
     List()           // конструктор
     {
          top=0;
          cout<<"\nkonstructor:\n";
     };
     ~List()          // деструктор
     {
          release();
          cout<<"\ndestructor:\n";
     }
     void push(int);  // добавление элемента
     void del()       // удаление элемента
     {
          Node* temp = top;
          top = top->next;
          delete temp;
     }
     void show();
     void release();// удаление всех элементов
};
void List::push(int i)
{
     Node* temp = new Node;
     temp->info = i;
     temp->next =top;
     top=temp;
}
void List::show()
{
     Node* temp=top;
     while (temp!=0)
     {
          cout<<temp->info<<"->";
```

```
            temp=temp->next;}
            cout<<endl;
        }


    void List::release()
    {
        while (top!=0)del();
    }
    int main()
    {
        List *p          ;//объявление указателя на List
        List st;//объявление объекта класса List
        int n = 0;
        cout << "Input an integer until 999: ";
        do      //добавление в список, пока не введено
999
        {
            cin >> n;
            st.push(n);
        } while(n != 999);
        st.show();
        st.del();
        p=&st;
        p->show();
        while(!kbhit());
        return 0;
    }
```

Необходимо отметить, что некоторый класс ClassA может использоваться до его объявления. В этом случае перед использованием класса ClassA нужно поместить его неполное объявление:

```
class ClassA;
```

Важнейшим принципом ООП является наследование. Класс, который наследуется, называется базовым, а наследуемый – производным. В следующем примере класс Derived наследует компоненты класса Base, точнее компоненты раздела public, которые остаются открытыми, и компоненты раздела protected (защищенный), которые остаются закрытыми. Компоненты раздела private, также наследуются, но являются не доступными напрямую для производного класса. Производному классу доступны все данные и методы базового класса, наследуемые из разделов public и protected. Для переопределенных методов в производном классе действует принцип полиморфизма, который будет рассмотрен ниже. Объекту базового класса можно присвоить объект

производного, указателю на базовый класс – значение указателя на производный класс. В этом случае через указатель на базовый класс можно получить доступ только к полям и функциям базового класса. Для доступа к полям и функциям порожденного класса следует привести (преобразовать) ссылку на базовый класс к ссылке на порожденный класс.

```
    #include <iostream.h>
    #include <conio.h>
    class Base                //   определение   базового
класса
    {
         int i;           //private по умолчанию
    protected:
         int k;
    public:
         Base()
         {
             i=0;
             k=1;
         }
         void set_i(int n);    // установка i
         int get_i()       // возврат i
         {
             return i;
         }
         void show()
         {
             cout<<i<<"  "<<k<<endl;
         }
    };                     //конец Base
    class Derived : public Base        //       производный
класс
    {
         int j;
    public:
         void set_j(int n);
         int mul();   //умножение i на к базового класса
                      //и на j производного
         }; //конец Derived
    //установка значения i в базовом классе

    void Base::set_i(int n)
    {
         i = n;
    }
```

```
    //установка значения j в производном классе

    void Derived::set_j(int n)
    {
         j = n;
    }

    //возврат i*k из Base умноженного на  j из Derived
    int Derived::mul()
    {
    /*производный класс наследует функции-члены базового
класса*/
         return  j  *  get_i()*k;//вызов  get_i()  базового
класса
    }
    int main()
    {
         Derived ob;
         ob.set_i(10);              //загрузка i в Base
         ob.set_j(4);          // загрузка j в Derived
         cout << ob.mul()<<endl;   //вывод числа 40
         ob.show();            //вывод i и k, 10 1
         Base bob=ob;          //присваивание       объекта
ссылке
                               //на базовой тип
         cout<<bob.get_i();         // вывод i
         cout.flush();
         while (!kbhit());
         return 0;
    }
```

Переменная i недоступна в производном классе, переменная k доступна, поскольку находится в разделе protected. В производном классе наследуются также функции get_i(), set_i() и show() класса Base из раздела public. Функция show() позволяет получить доступ из производного класса к закрытой переменной i производного класса. В результате выводится 40 10 1 10.

Принцип полиморфизма, состоящий в перегрузке методов, объявленных в различных классах с одним и тем же именем и списком параметров, будет рассмотрен в следующих темах.

Практическое задание №1

Цель работы: Изучить определение классов в языке C++.

Постановка задачи: Создать класс, отражающий структуру данных согласно выбранному варианту задания. Класс должен содержать методы для чтения, установки и отображения своих данных. Реализовать класс List для управления динамическим списков из элементов созданного класса.

Варианты заданий

№ варианта	Структура данных
1	Дата
2	Время
3	Книга
4	Ф.И.О.
5	Адрес
6	Студент
7	Автомобиль
8	Компьютер
9	Предприятие
0	Страна

Практическое занятие 2. Классы и функции

При реализации класса используются функции-члены класса (методы класса), функции-“друзья” класса, конструкторы, деструкторы, функции-операторы. Функции-члены класса объявляются внутри класса. Определение функции обычно помещается вне класса. При этом перед именем функции помещается операция доступа к области видимости имя_класса::. Таким образом, определение функции-члена класса имеет вид:

```
тип имя_класса:: имя_функции (список аргументов) {  }
```

Определение функции может находиться и внутри класса вслед за его объявлением. Такие функции называются inline-функциями. Рассмотрим пример:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#define SIZE 255
class Strtype
{
     char *p;
     int len;
public:
     Strtype()    //  инициализация  объекта,  inline
конструктор
     {
          p=new char [SIZE];
          if(!p)
          {
               cout << "ошибка выделения памяти\n";
               exit(1);
          }
          *p='\0';
          len=0;
     }
     ~Strtype(); //деструктор
     void set(char *ptr);
     char *get()
     {
          return p;
     }
     void show();
```

```
};
// освобождение памяти при удалении объекта строка
inline Strtype::~Strtype()//inline-деструктор
{
     cout << "освобождение p\n";
     delete [] p;
}
void Strtype::set(char *ptr)
{
     if(strlen(ptr) >= SIZE)
     {
          cout << "строка слишком велика \n";
          return;
     }
     strcpy(p, ptr);
     len = strlen(p);
}
void Strtype::show()
{
     cout << p << " длина : " << len<< "\n";
}
int main()
{
 {   Strtype s1,s2;
     s1.set("This is a test");
     s2.set("Second test");
     s1.show();
     s2.show();
     cout<<s2.get()<<endl;
 }
     while (!kbhit());
     return 0;
}
```

В результате будет выведено:

```
This is a test длина: 14
Second test длина: 11
Second test
Освобождение p
Освобождение p
```

Конструктор Strtype() и функция get() являются inline-функциями. Деструктор ~Strtype() также является inline-функцией, хотя и определяется вне класса. Ключевое слово inline содержит указание компилятору создать код

функции, подставляемый в точку вызова. При этом время вызова сокращается, хотя код может увеличиться.

Вызов функций-членов класса осуществляется одним из двух способов:

- имя_объекта.имя_функции(аргументы);
- указатель_на_объект -> имя_функции(аргументы);

Еще раз обращаем внимание на то, что при определении функции- члена класса перед именем функции стоит имя класса, а при вызове –имя объекта класса.

Указатель this

Обращение к данным класса осуществляется с помощью функций, которые могут вызываться различными объектами класса. Возникает вопрос: откуда функция “знает”, какой объект его вызвал и какие данные при этом должны быть использованы? Например, в предыдущем примере вызовы функции s1.show() и s2.show() приводят к выводу различных для s1и s2 значений строк p. Это происходит вследствие того, что каждый объект имеет скрытый указатель this на самого себя, объявляемый неявно. Значением этого указателя, который автоматически передается функциям-членам класса при их вызове, является адрес начала объекта. В результате функция show() в реальности содержит инструкцию:

```
cout << this->p << " длина : " << this->len<< "\n";
```

При этом *this представляет сам объект, this->имя_члена – ссылка на данные объекта класса. Рассмотрим пример:

```
#include<iostream.h>
#include<conio.h>
class MyClass
{
     int x,y;
     public:
     void set(int xx, int yy)      //устанавливает
данные
     {
          x=xx;    //равносильно this->x=xx;this->y=yy;
          y=yy;
     }
     MyClass f(MyClass &); //возвращает объект
     MyClass *ff()//возвращает указатель на вызвавший
объект
     {
          x=y=100;
          return this;
```

```
        }
        void display()
        {
            cout<<x<<'\t'<<y<<endl;
        }
};
MyClass MyClass::f(MyClass &M)
{
        x+=M.x;
        y+=M.y;
        return *this;
}
int main()
{
        MyClass a, b;
        a.set(10,20);
        b.ff()->display();   //результат 100,100
        b.display();     //результат 100,100
        a.display();     //результат 10,20
        a.f(b).display();    //результат 110,120
        a.display();     //результат 110,120
        while (!kbhit());
        return 0;
}
```

Так как функция b.ff() возвращает указатель this на объект b, то можно вызвать функцию display() как this->display(). При этом выводятся поля данных объекта b(100,100). При вызове a.f(b).display() функция f(b) возвращает значение вызвавшего ее объекта a как *this. Затем вызывается функция display() как a.display().

Отметим, что в функцию лучше передавать не значение объекта, а ссылку на него. Это дает экономию стековой памяти, поскольку объект не копируется, и более безопасно для объектов, использующих динамическую память. Уничтожение копий аргументов при выходе из функции не может разрушить такие объекты при освобождении динамической памяти.

Функции-друзья класса, объявляемые со спецификатором friend, указатель this не содержат. Объекты, с которыми работают такие функции, должны передаваться в качестве их аргументов.

Конструкторы и деструкторы

Конструктор – это функция-член класса, которая вызывается автоматически для создания и инициализации экземпляра класса. Известно, что экземпляры структур можно инициализировать при объявлении:

```
struct Student
{
     int semhours;          //public по умолчанию
     char subj;
}
Student s={0,”c” };         //объявление                  и
инициализация
```

При использовании класса приложение не имеет доступа к его защищенным элементам, поэтому для класса подобную инициализацию выполнить нельзя. Для этой цели используются специальная функция-член класса, инициализирующая экземпляр класса и указанные переменные при создании объекта. Подобная функция вызывается всегда при создании объекта класса. Это и есть конструктор – функция-член класса, которая имеет то же имя, что и класс. Конструктор может быть подставляемой (inline) и неподставляемой функциями. Рассмотрим пример:

```
#include<iostream.h>
class Student
{
     int semhours;
     char subj;
public:
     Student() //inline-конструктор 1 без параметров
     {
          semhours=0;
          subj='A';
     }
     Student(int, char);//объявление
                        //конструктора 2 с параметрами
};
Student::Student(int hours,char g)//конструктор 2
{
     semhours=hours;
     subj=g;
}
int main()
{
     Student s(100,'A'); //конструктор 2
     Student s1[5]; //конструктор1 вызывается 5 раз
     return 0;
}
```

Там, где находятся объявления s и s1, компилятор помещает вызов соответствующего конструктора Student().

Конструктор не имеет типа возвращаемого значения, хотя может иметь аргументы и быть перегружаемым. Он вызывается автоматически при создании объекта, выполнении оператора new или копировании объекта. Если конструктор отсутствует в классе, компилятор C++ генерирует конструктор по умолчанию.

Деструктор вызывается автоматически при уничтожении объекта, и имеет то же имя, что и класс, но перед ним стоит символ “~”. Деструктор можно вызывать явно в отличие от конструктора. Конструкторы и деструкторы не наследуются, хотя производный класс может вызывать конструктор базового класса.

В следующем примере используется конструктор копирования, который необходимо вводить из-за проблемы, связанной с динамической памятью. Когда объект передается в функцию по значению, то создается его копия, при этом конструктор не вызывается. При выходе из функции объект уничтожается и вызывается деструктор, освобождающий его динамическую память. В этом случае исходный объект, использующий ту же динамическую память, что и копия, может быть поврежден. Аналогично, если функция возвращает объект, содержащий динамические переменные, при выходе из функции копия этого объекта разрушается вызовом деструктора. Возвращаемый объект при этом также может быть разрушен. Равным образом это повторяется и при инициализации объявляемого объекта вида:

```
Class_type B=A;
```

При этом происходит побитовое копирование объекта A в объект B. Однако для массива в объекте B память динамически не выделяется, а происходит только копирование указателя на массив. Если затем объект A, содержащий динамический массив, разрушается вызовом деструктора, то объект B также разрушается. Поэтому при таком объявлении должен вызваться конструктор копирования.

Общая форма конструктора копирования имеет вид:

```
имя_класса (const имя_класса &ob)
{
/*выделение памяти и копирование в нее ob*/
}
```

Здесь ob является ссылкой на объект в правой части инициализации.

Рассмотрим пример использования конструктора копирования при создании безопасного динамического массива.

```
/*создается класс "безопасный" массив. Когда один
объект-массив используется для инициализации другого, для
выделения памяти вызывается конструктор копирования*/
```

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
class Array
{
     int *p;
     int size;
public:
     Array (int s)
     {
          cout << "constructor1 \n";
          size=s;
          p=new int[size];
          if(!p) exit(1);
     }
     ~Array()
     {
          delete [ ] p;
     }
     Array(const Array &a);    //конструктор
копирования
     void put(int i, int j)
     {
          if(i>=0 && i<size) p[i]=j;
     }
     int get(int i)
     {
          return p[i];
     }
};
/*  конструктор  копирования.  Память  выделяется  для
копии,  адрес  памяти передается p.*/
Array:: Array(const Array &a)
{
     p=new int[a.size];   //выделение    памяти    для
копии
     if(!p) exit(1);
     for(int i=0;i<a.size;i++)
     p[i]=a.p[i];//копирование  содержимого  в  память
для копии
     cout << "constructorcopy2\n";
}
int main()
{
     Array y(5);      //вызов обычного конструктора
```

```
        for(int i=0;i<5;i++)
        {   //помещение в массив нескольких значений
            y.put(i, i+5);
            cout << y.get(i);    //вывод на экран num
        }
        cout << "\n";
        Array x=y;
    /*создание   массива   x   и   инициализация,   вызов
конструктора копирования */
    // другой способ вызова - объявление: Array x(y);
        y=x;//конструктор не вызывается !!!
        while(!kbhit());
        return 0;
    }
```

Результат:

```
constructor1
56789
constructorcpy2
```

Объект y используется при инициализации объекта x с помощью конструктора копирования. При этом выделяется динамическая память, адрес которой в x.p, и производится копирование y в x. После этого y и x не разделяют одну и ту же динамическую память. Если же присваивание осуществляется не при инициализации, то конструктор копирования не вызывается, а происходит побитовое копирование объекта. В результате двумя объектами используется одна и та же динамическая память. Так происходит в случае присваивания y=x в примере.

В следующем примере конструктор копирования вызывается при передаче объекта-строки в качестве аргумента функции show():

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
class Strtype
{
     char *p;
     int l;
public:
     Strtype(char *s='\0')     //конструктор
     {
         l=strlen(s);
         p=new char[l+1];
         if(!p)
```

```
            {
                cout<<     "ошибка     при     выделении
памяти\n";exit(1);
            }
            strcpy(p, s);
            cout<<"constr1\n";
    }
        Strtype(const      Strtype      &o)//конструктор
копирования
        {
            l=strlen(o.p);
            p=new char[l+1];//выделение памяти для новой
копии
            if(!p)
            {
                cout << "ошибка памяти\n";
                exit(1);
            }
            strcpy(p, o.p);  //передача строки в копию
            cout<<"constrcpy\n";
        }
        ~Strtype()       //деструктор
        {
            delete [ ] p;
            cout<<"destr\n";
        }
        char *get()
        {
            return p;
        }
    };
    void show(Strtype x)
    {
        char *s;
        s=x.get();
        cout << s << "\n";
    }
    int main()
    {
        Strtype a("Hello"), b("There");//constr1,constr1
        show(a);//конструктор                  копирования
constrcpy,Hello,destr
        show(b);//конструктор                  копирования
constrcpy,There,destr
        while(!kbhit());
```

```
        return 0;
    }
```

Когда функция show() завершается и x выходит из области видимости, то освобождается память x.p, однако не та, которая используется передаваемым в функцию объектом.

Организация внешнего доступа к компонентам класса. Функции "друзья" класса.

В языке C++ одна и та же функция не может быть компонентом двух разных классов. Чтобы предоставить функции возможность выполнения действий над различными классами можно определить обычную функцию языка C++ и предоставить ей право доступа к элементам класса типа private, protected. Для этого нужно в описании класса поместить заголовок функции, перед которым поставить ключевое слово friend. Рассмотрим пример:

```
    //функция sameColor() сравнивает цвет прямоугольника
и линии
    class Line;//неполное объявление класса
    class Box
    {
        int color;        // цвет прямоугольника
        int upx, upy;          // левый верхний угол
        int lowx, lowy;  //правый нижний угол
    public:
        // объявление friend-функции
        friend int sameColor(line l, box b);
        void setColor(int c);
        void defBox(int x1, int y1, int x2, int y2);
    };
    class Line
    {
        int color;
        int startx, starty;
        int endx,endy;
    public:
        friend int sameColor(Line l, Box b);
        void setColor(int c);
        void defLine(int x, int y, int l);
    };
    int sameColor(Line l, Box b)
    {
        if(l.color==b.color) return 1;
        return 0;
```

```
}
```

Если не использовать friend-функцию, то необходимо сначала вернуть цвета объектов, а затем написать функцию их сравнения. В приведенном примере класс Box использует класс Line, объявленный позже. Поэтому перед объявлением класса Box ставится предварительное неполное объявление вида:

```
class имя_класса;
```

Хотя дружественная функция “знает” элементы класса, доступ к ним возможен только через объект класса, т.к. указатель this ей не передается. В следующем примере дружественная функция использует указатель на класс в качестве параметра и для доступа к элементам i и k через указатель. Прямой доступ к ним невозможен.

```
class X
{
     int i,k;
public:
     void memFunc(int);
     friend void frFunc(X*, int);
};
void X::memFunc(int a)
{
     i=a;     //прямой доступ
     k=a;
}
void fr_func(x *xptr, int a)
{
//доступ к i и k через указатель, ошибка, если
записать к=а
     xptr->i=a;
     xptr->k=a;
}
int main()
{
     X xobj;//объявление объекта
     frFunc(&xobj, 6);//вызов дружественной функции
     xobj.memFunc(6);//вызов члена класса
     return 0;
}
```

Разрешается объявлять функции-члены класса дружественными для другого класса. В следующем примере функция memF() член класса X получает доступ к защищенным полям класса Y.

```
class Y;
class X
{//...
     void memF(Y ob);
};
class Y
{//…
     friend void X::memF(Y ob);
};
```

Операторы-функции

Операторы-функции используются для введения операций, связанных с символами:

```
+ , - , * , / , % , ^ , & ,  | , ~ , ! , = , < , > ,+= , [ ] , -> , ( ) , new, delete.
```

Оператор-функция является членом класса или дружественной классу. Общая форма оператора-функции члена класса:

```
возвращаемый_тип имя_класса::operator#(список_аргум)
{ /*выполняемые действия */ }
```

После этого вместо operator#(a,b) можно писать a#b. Здесь символ # представляет один из введенных выше символов. В качестве примера можно привести операции >>, <<, перегружаемые для ввода-вывода. Отметим, что при перегрузке нельзя менять приоритет операторов и число операндов. Если оператор-функция, являющаяся членом класса перегружает бинарный оператор, у функции будет только один параметр-объект, находящийся справа от знака оператора. Объект слева вызывает оператор-функцию и передается неявно с помощью указателя this. Например:

```
// перегрузка  +, = и ++ для класса  coord.
#include <iostream.h>
#include <conio.h>
class Coord
{
     int x, y; // значения координат
public:
     Coord(int i=0, int j=0) { x = i; y = j; }
     void getXY(int &i, int &j) { i = x; j = y; }
     Coord operator+(Coord ob2);
     Coord operator=(Coord ob2);
     Coord operator++(); // префиксная форма
```

```
};
Coord Coord::operator+(Coord ob2) //перегрузка  +
{
     Coord temp;
     temp.x = x + ob2.x; // temp.x=this->x+ob2.x
     temp.y = y + ob2.y; // temp.y=this->y+ob2.y
     return temp;
}
Coord Coord::operator=(Coord ob2) //перегрузка =
{
     x = ob2.x;   // this->x=ob2.x
     y = ob2.y;   // this->y=ob2.y
     return *this;    //возвращение объекта,
                  //которому присвоено значение
}
Coord  Coord::operator++()  //перегрузка  ++,  унарный
оператор
{
     x++;
     y++;
     return *this;
}
int main()
{
     Coord o1(10, 10), o2(5, 3), o3;
     int x, y;
     o3  =  o1  +  o2;  //сложение  двух  объектов.  Вызов
operator+()
     o3.getXY(x, y);
     cout << "(o1 + o2) X: " << x << ", Y: " << y <<
"\n";
     o3 = o1; //присваивание объектов
     o3.getXY(x, y);
     cout << "(o3 = o1) X: " << x << ", Y: " << y <<
"\n";
     ++o1; //инкрементация объекта
     o1.getXY(x, y);
     cout << "(++o1 )     X: " << x << ", Y: " << y <<
"\n";
     while(!kbhit());
     return 0;
}
```

Результат:

```
 (o1+o2) X:15 ,y:13
 (o3=o1) X:10, y:10
```

```
 (++o1)  X:11, Y:11
```

При перегрузке унарного оператора ++ единственный параметр-объект передается через указатель this.

В следующем примере вводятся класс “множество” и операции && – пересечения, << – вывод множества.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class Set
{
     char *pl; //указатель на множество элементов типа char
public:
     Set(char *pi)//конструктор
     {
          pi=new char[strlen(pl)+1];
          strcpy(pi,pl);
     }
     Set &operator &&(Set &); //перегрузка &&-пересечение
     //перегрузка<<
     friend ostream &operator<<(ostream &stream,Set &ob);
     ~Set(){delete [] pi;}//деструктор
};
Set& Set::operator &&(Set &s) // пересечение A=A^B
{
     int  l=0;
     for (int j=0;pi[j]!=0;j++)
          for (int k=0;s.pi[k]!=0;k++)
               if (pi[j]==s.pi[k])
               {
                    pi[l]=pi[j];
                    l++;
                    break;
               }
     pi[l]=0;
     return *this;
}
ostream &operator<<(ostream &stream, Set &ob)
{
     stream << ob.pi << '\n'; /*перегрузка вывода */
     return stream;
}
```

```
int main()
{
      Set s1("1f2bg5e6"),s2("abcdef");
      Set s3=s2;
      cout<<(s1&&s2)<<endl;//результат fbe
      cout<<s3<<endl;//результат abcdef
      while(!kbhit());
      return 0;
}
```

Оператор присваивания

Когда одному объекту присваивается значение другого, происходит его копирование. Если побитовое копирование нежелательно из-за того, что копии объектов ссылаются на одну и ту же динамическую память, можно использовать собственную функцию копирования operator=(). Рассмотрим пример:

```
Strtype &Strtype::operator=(const Strtype &ob)
{
      if(&ob==this) return *this;
      if(l < ob.l)
      { //требуется выделение дополнительной памяти
          delete[ ] p;
          p = new char [ob.l+1];
          if(!p)
          {
               cout << "ошибка выделения памяти \n";
               exit(1);
          }
      }
      l = ob.l;
      strcpy(p, ob.p);
      return *this;
}
```

Здесь operator=() возвращает ссылку на объект. Параметром также является ссылка, что запрещает создание копии объекта, стоящего справа от операции присваивания. Добавление этого оператора в класс позволяет копировать объекты с помощью оператора присваивания, который не наследуется и должен быть членом класса.

Использование дружественных операторов-функций

В дружественную функцию не передается указатель this, поэтому при перегрузке для унарного оператора в оператор-функцию передается один параметр, для бинарного – два. Оператор присваивания не может быть дружественной функцией, а только членом класса. В следующем примере рассматривается использование дружественных оператор-функций +, *, /, <<, >> для класса Complex.

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
class Complex
{
     double re, im;
public:
     Complex (double  r=0,double i=0)//конструктор
     {
          re=r;
          im=i;
     }
     friend Complex operator+(Complex, Complex);
     friend Complex operator*(Complex, Complex);
     friend Complex operator/(Complex, Complex);
     friend  ostream  &  operator<<(ostream  &stream,
Complex ob);
     friend  istream  &  operator>>(istream  &stream,
Complex ob);
};
     ostream &operator<<(ostream &stream, Complex ob)
     {
          stream  <<"("<<  ob.re  <<  ",  "  <<  ob.im
<<")"<< '\n';
          return stream;
     }
istream &operator>>(istream &stream, Complex &ob)
{
     stream >> ob.re >> ob.im;
     return stream;
}
Complex operator+(Complex a1, Complex a2)
{
     return Complex(a1.re+a2.re,a1.im+a2.im);
}
Complex operator*(Complex a1, Complex a2)
{
     return Complex(a1.re*a2.re-a1.im*a2.im,
                    a1.re*a2.im+a1.im*a2.re);
```

```
}
Complex operator/(Complex a1, Complex a2)
{
      double r=a2.re,i=a2.im;
      double tr=r*r+i*i;
      return Complex((a1.re*r+a2.im*i)/tr,
                  (a1.im*r-a1.re*i)/tr);
}
int main()
{
      Complex a(1.,2.),b(2.,2.),c;
      c=a+b;
      cout<<c<<a*b<<a/b<<endl;
      cout<<"Enter complex number:";
      cin>>c;
      cout<<c;
      while(!kbhit());
      return 0;
}
```

Практическое задание №2

Цель работы: Научиться работать с функциями-членами класса, конструкторами, деструкторами, дружественными классами и перегруженными операторами.

Постановка задачи: Создать класс согласно выбранному варианту задания. Реализовать процедуры ввода и отображения данных.

Варианты заданий

№ варианта	Задание
1	Однонаправленный список целых чисел с использованием inline-функций.
2	Стек из строк с использованием inline конструкторов и деструкторов.
3	Однонаправленный список объектов типа «Книга» с использованием указателя this.
4	Стек из объектов типа «Адрес» с использованием указателя this.
5	Создать «безопасный массив» из объектов типа «Автомобиль» с использованием конструкторов и деструкторов.
6	Однонаправленный список объектов типа «Дата» с использованием конструкторов копирования.
7	Создать объекты типа «Вектор» и «Матрица». Реализовать в одном их этих классов дружественную функции для умножения вектора на матрицу.
8	Однонаправленный список из объектов типа «Дата». Для класса «Дата» перегрузить операторы сложения и вычитания.
9	Однонаправленный список из объектов типа «Время». Для класса «Время» перегрузить оператор присваивания.
0	Для классов «Вектор» и «Матрица» перегрузить операции сложения, вычитания и умножения с использованием дружественных операторов-функций.

Практическое занятие 3. Наследование и полиморфизм

При реализации наследования производному классу (потомку) передаются свойства базового класса, объявленные в разделах public и protected. Члены раздела protected являются частными для базового и производного классов. При наследовании класса может объявляться спецификатор доступа к членам базового класса.

```
class имя_класса: доступ имя_базового_класса
{
//члены класса
}
```

Спецификатор доступа при наследовании может принимать одно из трех значений: public (по умолчанию при наследовании структур), private (по умолчанию при наследовании классов) и protected. В случае принятия спецификатором значения public все члены разделов public и protected базового класса становятся членами разделов public и protected производного класса. Члены раздела private (частные) недоступны для производного класса. Если доступ имеет значение private, то все члены разделов public и protected становятся членами раздела private производного класса. Когда же доступ принимает значение protected, то все члены разделов public, protected переходят в раздел protected производного класса. Рассмотрим пример:

```
#include <iostream.h>
#include <conio.h>
class X
{
protected: //обязательно для наследования
     int i, j;
public:
     void getIJ()
     {
          cout << "Enter two number: ";
          cin >> i >> j;
     }
     void showIJ()
     {
          cout << i << " " << j << "\n";
     }
};
class Y : public X
{
     /* в классе Y, i и j - защищенные данные из X */
     int k;
```

```
    public:
         int getK()
         {
             return k;
         }
         void makeK()
         {
             k = i*j;
         }
    };
    /*Z имеет доступ к i и j из X, но не к частному  k
класса Y */
    class Z : public Y
    {
    public:
         void f()
         {
             i = 2;
             j = 3;
         }// i и j доступны отсюда
    };
    int main()
    {
         Y v1;
         Z v2;
         v1.getIJ();
         v1.showIJ();
         v1.makeK();
         cout << v1.getK()<<"\n";
         v2.f();
         v2.showIJ();
         while(!kbhit());
         return 0;
    }
```
Вывод:
```
Enter two numbers:5 6
         5 6
         30
         2 3
```

Конструкторы и деструкторы производных классов

Конструкторы и деструкторы базовых классов не наследуются производными классами, однако они вызываются при создании объектов этих классов. При этом конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке. Для передачи аргументов конструктору

базового класса при использовании производного класса применяется следующий синтаксис:

```
конструктор_производного_класса(арг):Base(арг)
{
     //тело конструктора производного класса
}
```

Рассмотрим пример:

```
#include <iostream.h>
#include <conio.h>
class Base
{
     int i;
public:
     Base(int n)
     {
          cout << "constructor1 \n";
          i = n;
     }
     ~Base()
     {
          cout<<"destructor1 \n";
     }
     void showi()
     {
          cout << i << '\n';
     }
};
class Derived : public Base
{
     int j;
public:
     Derived(int n):Base(n+5)
     /*передача аргумента в базовый класс */
     {
          cout << "constructor2  \n";
          j = n;
     }
     ~Derived()
     {
          cout<<"destructor2\n";
     }
     void showj()
```

```
        {
            cout << j << '\n';
        }
    };
    int main()
    {
        {
            Derived o(3);
            /*фигурные скобки добавлены, чтобы вывести
сообщения              деструкторов */
            o.showi();
            o.showj();
        }
        while(!kbhit());
        return 0;
    }
```

Вывод:

```
constructor1
constructor2
8
3
destructor2
destructor1
```

Множественное наследование

Один класс может наследовать атрибуты двух и более базовых классов, которые перечисляются после двоеточия через запятую. Если базовые классы содержат конструкторы, то они вызываются поочередно в порядке перечисления.

```
#include <iostream.h>
#include <conio.h>
class X
{
protected:
     int a;
public:
     X()
     {
          a = 10;
          cout << "инициализация X\n";
     }
     ~X()
     {
          cout << "деинициализация X\n";
```

```
      }
};
class Y
{
protected:
      int b;
public:
      Y()
      {
            cout << "инициализация Y\n";
            b = 20;
      }
      ~Y()
      {
            cout << "деинициализация Y\n";
      }
};
// Z наследует как  X, так и Y
class Z : public X, public Y
{
public:
      Z()
      {
            cout << "инициализация Z\n";
      }
      ~Z()
      {
            cout << "деинициализация Z\n";
      }
      int make_ab() { return a*b; }
};
int main()
{
      {
            Z ob;
            cout << ob.make_ab();
      }
      while(!kbhit());
      return 0;
}
```

Результат:

```
инициализация X
инициализация Y
инициализация Z
200
```

```
деинициализация Z
деинициализация Y
деинициализация X
```

В C++ указатели и ссылки на базовый класс могут быть использованы для ссылок на объект производного класса. Если параметр функции является ссылкой на базовый класс, то аргументом функции может быть как объект базового класса, так и объект производного. Однако через указатель или ссылку базового класса, которые в действительности ссылаются на объект производного класса, можно получить доступ только к тем объектам (полям и функциям) производного класса, которые были унаследованы от базового класса. В то же время указатель или ссылку производного класса нельзя использовать для доступа к объекту базового класса. В этом случае необходимо применить явное преобразование типов, используя операцию (тип).

Виртуальные функции и полиморфизм

Механизм виртуальных функций в ООП используется для реализации полиморфизма: создания метода, предназначенного для работы с различными объектами из иерархии наследования за счет механизма позднего (динамического) связывания. Виртуальные функции объявляются в базовом и производном классах с ключевым словом virtual, имеют одинаковое имя, список аргументов и возвращаемое значение. Метод, объявленный в базовом классе как виртуальный, будет виртуальным во всех производных классах, где он встретится, даже если слово virtual отсутствует.

В отличие от механизма перегрузки функций (функции с одним и тем же именем, но с различными типами аргументов считаются разными) виртуальные функции объявляются в порожденных классах с тем же именем, возвращаемым значением и типом аргументов. Если различны типы аргументов, виртуальный механизм игнорируется. Тип возвращаемого значения переопределить нельзя.

Основная идея в использовании виртуальных функций состоит в следующем: такая функция может быть объявлена в базовом классе, а затем переопределена в каждом производном классе. Каждый объект класса, управляемого из базового класса с виртуальными функциями, содержит указатель на таблицу vmtbl (virtual method table) с адресами виртуальных функций. Эти адреса устанавливаются в адреса нужных для данного объекта функций во время выполнения. При этом доступ через указатель на объект базового класса осуществляется к виртуальной функции из базового класса, а доступ через указатель на объект производного класса – на функцию из этого же класса. То же происходит при передаче функции объекта производного класса, если аргумент объявлен как базовый класс.

Оба варианта рассмотрены в следующем примере:

```
/*здесь указатель на базовый класс и ссылка используются для доступа к виртуальной функции */
#include <iostream.h>
```

```
#include <conio.h>
class Base
{
public:
     virtual void who()   //  определение  виртуальной
функции
     {
          cout << "Base\n";
     }
};
class First : public Base
{
public:
     void who()   //  определение  who()  относительно
First
     {
          cout << "First\n";
     }
};
class Second : public Base
{
public:
     void  who()//  определение  who()  относительно
Second
     {
          cout << "Second\n";
     }
};
/* параметр ссылка на объект базового класса */
void show_who(Base &r)
{
     r.who();
}
int main()
{
     Base base_obj;
     Base* pb;
     pb=&base_obj;
     pb->who();       //base_obj.who()
     First first_obj;
     pb=&first_obj;
     pb->who();    //first_obj.who()
     Second second_obj;
     pb=&second_obj ;
     pb->who();         //second_obj.who()
```

```
     show_who(base_obj); // доступ к Base's who()
     show_who(first_obj); // доступ к First's who()
     show_who(second_obj); // доступ к Second's who()
     while(!kbhit());
     return 0;
}
```

Вывод:
```
Base
First
Second
Base
First
Second
```

В следующем примере рассматривается реализация стека и очереди на основе односвязного списка, при этом очередь и стек являются потомками класса List:

```
/*классы, наследование и виртуальные функции
создание класса родовой список для целых*/
#include <iostream.h>
#include <stdlib.h>
class List
{
public:
     List *head; //указатель на начало списка
     List *tail;//указатель на конец списка
     List *next;//указатель на следующий элемент
     int num;          //число для хранения
     List ()
     {
          head = tail = next = NULL;
     }
     /*абстрактная базовая виртуальная функция*/
     virtual void store(int i) = 0;
     /*абстрактная базовая виртуальная функция*/
     virtual int retrieve() = 0;
 };
 //создание типа очередь на базе списка
class Queue : public List
{
public:
     void store(int i);
     int retrieve();
```

```
     queue operator+(int i)
     {
          store(i);
          return *this;
     }
     /* перегрузка постфиксного инкремента */
     int operator --(int unused)
     {
          return retrieve();
     }
};
void Queue::store(int i)
{
     List *item;
     item = new Queue;
     if(!item)
     {
          cout << "ошибка выделения памяти\n";
          exit(1);
     }
     item -> num = i;
     //добавление в конец списка
     if(tail) tail -> next = item;
     tail = item;
     item -> next = NULL;
     if(!head) head = tail;
}
int Queue::retrieve()
{
     int i;
     List *p;
     if(!head) {cout << "список пуст\n";return 0;   }
     //удаление из начала списка
     i = head -> num;
     p = head;
     head = head -> next;
     delete p;
     return i;
}
class Stack : public List
{/*создание класса стек на базе списка */
public:
     void store(int i);
     int retrieve();
     Stack operator+(int i)
```

```
    {
        store(i);
        return *this;
    }
    int operator --(int unused)
    {
        return retrieve();
    }
};
void Stack::store(int i)
{
    List *item;
    item = new Stack;
    if(!item) {
        cout << "ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;
    //добавление в начало списка, как в стеке
    if(head) item -> next = head;
    head = item;
    if(!tail) tail = head;
}
int Stack::retrieve()
{
    int i;
    List *p;
    if(!head)
    {
        cout << "список пуст\n";
        return 0;
    }
    //удаление из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;
    return i;
 }
int main()
{
    List *p;
    //демонстрация очереди
    Queue q_ob;
    p = &q_ob; //указывает на очередь
```

```
        q_ob + 1;
        q_ob + 2;
        q_ob + 3;
        cout << "очередь : ";
        cout << q_ob --;// вывод 1
        cout << q_ob --;// вывод 2
        cout << q_ob --;// вывод 3
        cout << '\n';
        //демострация стека
        Stack s_ob;
        p = &s_ob; //указывает на стек
        s_ob + 1;
        s_ob + 2;
        s_ob + 3;
        cout << "стек: ";
        cout << s_ob --;// вывод 3
        cout << s_ob --;// вывод 2
        cout << s_ob --;// вывод 1
        cout << '\n';
        return 0;
 }
```

Если конструкторы не могут быть виртуальными, деструкторы могут (для полиморфных объектов их рекомендуется объявлять в базовых классах как виртуальные).

Абстрактные классы и чисто виртуальные функции

Абстрактные классы – это классы, содержащие чисто виртуальные функции, которые при объявлении в классе приравниваются к нулю. Абстрактные классы используются только для наследования, так как объекты таких классов не могут быть созданы.

```
#include<iostream.h>
class Shape       //абстрактный
{
     int xb,yb,xc,yc;
public:
     Shape(int hb,int vb,int hc,int vc)://конструктор
     /* применяется список инициализаторов */
     xb(hb),yb(vb),xc(hc),yc(vc){}
     virtual void move(int,int)=0;//чисто виртуальная
функция
```

```
        virtual     void     copy(int,int)=0;//     чисто
виртуальная функция
        virtual void rotate(int)=0;// чисто виртуальная
функция
    };
```

При реализации конструктора использован список инициализаторов, который помещается после двоеточия следующего за заголовком конструктора. Для каждого поля в скобках указывается инициализирующее значение (может быть выражением). Это единственный способ инициализации полей констант, полей ссылок и полей объектов, у которых есть только конструкторы с параметрами. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

Абстрактные базовые классы используются для создания общего для множества иерархических классов интерфейса. В следующем примере рассматриваются два класса с общим интерфейсом, через который осуществляется вызов функций с помощью указателей на соответствующую vtbl – таблицу виртуальных методов.

```
    #include <iostream.h>
    #define interface struct
    interface IX
    {
         virtual void _stdcall FX1()=0;
         virtual void _stdcall FX2()=0;
    };
    class CA:public IX
    {
         /*наследование структуры  */
    public:
         virtual               void               _stdcall
FX1(){cout<<"CA::FX1"<<endl;}
         virtual               void               _stdcall
FX2(){cout<<"CA::FX2"<<endl;}
    };
    class CB:public IX
    {
    public:
         virtual               void               _stdcall
FX1(){cout<<"CB::FX1"<<endl;}
         virtual               void               _stdcall
FX2(){cout<<"CB::FX2"<<endl;}
    };
    void ff(IX* pix)
    {
```

```
     pix->FX1();
     pix->FX2();
}
int main()
{
     CA* pA=new CA;//создание экземпляра CA
     CB* pB=new CB;//создание экземпляра CB
     IX* pix=pA;
     ff(pix);//вызов методов CA
     pix=pB;
     ff(pix);//вызов методов CB
     return 0;
}
```

Практическое задание №3

Цель работы: Изучить объектно-ориентированные концепции наследования и полиморфизма в языке программирования C++.

Постановка задачи: Создать набор классов согласно выбранному варианту задания. Реализовать процедуры ввода и отображения данных.

Варианты заданий

№ варианта	Задание
1	Создать базовый класс «Фигура» и несколько производных от него классов: «Точка», «Квадрат», «Круг» и т.д. Классы должны содержать приватные, защищенные и общедоступные члены данных.
2	Создать базовый класс «Транспорт» и несколько производных от него классов: «Автомобиль», «Самолет», «Поезд» и т.д. Классы должны содержать приватные, защищенные и общедоступные члены данных.
3	Создать базовый класс «Фигура» с использованием конструктора с аргументами и деструктора. Создать несколько производных от него классов: «Точка», «Квадрат», «Круг» и т.д.
4	Создать базовый класс «Транспорт» с использованием конструктора с аргументами и деструктора. Создать несколько производных от него классов: «Автомобиль», «Самолет», «Поезд» и т.д.
5	Создать базовый класс «Фигура» и производные от него классы «Точка», «Квадрат», «Круг». Создать базовый класс «Оформление» и производные от него классы «Цвет» и «Стиль». На основе этих классов создать набор различных классов с использованием множественного наследования.
6	Создать базовый класс «Транспорт» и производные от него классы «Автомобиль», «Поезд», «Самолет». Создать базовый класс «Тип» и производные от него классы «Грузовой» и «Пассажирский». На основе этих классов создать набор различных классов с использованием множественного наследования.
7	Выполнить задание 1 с использованием виртуальных

	функций для методов ввода и вывода данных.
8	Выполнить задание 2 с использованием виртуальных функций для методов ввода и вывода данных.
9	Выполнить задание 1, при условии, что базовый класс должен быть абстрактным.
0	Выполнить задание 2, при условии, что базовый класс должен быть абстрактным.

Практическое занятие 4. Массивы объектов, указатели и ссылки

Массивы объектов создаются так же, как и массивы переменных. Если класс содержит конструктор, массив может быть инициализирован, причем конструктор вызывается столько раз, сколько элементов содержит массив.

```
#include<iostream.h>
class Sample
{
     int a;
public:
     Sample(int n)
     {
          a=n;
          cout<<”constructor\n”;
     }
     int getA(){return a;}
};
int main()
{
     Sample ob[4]={1,2,3,4};
     Sample *pob=ob;
     int i;
     for(i=0;i<4;i++)
          cout<<ob[i].getA ()<< ‘ ‘;
     cout<<”\n”;
     cout<<pob->getA ();
     return 0;
}
```

Программа выведет 1, 2, 3, 4, конструктор вызывается четыре раза. Затем еще раз будет выведено 1 с помощью указателя pob на первый элемент массива. Список инициализации – это сокращение общей конструкции:

```
    Sample
ob[4]={Sample(1),Sample(2),Sample(3),Sample(4)};
```

Такая конструкция становится основной, если конструктор имеет два и более аргумента. Например:

```
Sample                                                ob[4]=
 {Sample(1,2),Sample(3,4),Sample(5,6),
              Sample(7,8)};
```

При создании динамических объектов используются указатели на объект оператор new, который вызывает конструктор и производит инициализацию. Для разрушения динамического объекта применяется оператор delete, который может помещаться в деструкторе. Например:

```
#include <iostream.h>
class Samp
{
     int i, j;
public:
     Samp()
     {
          cout<<”конструктор2\n”;
     }
     Samp(int a,int b)
     {
          i=a;j=b;
          cout<<”конструктор1\n”;
     }
     void setIJ(int a, int b)
     {
          i = a;
          j = b;
     }
     ~Samp() { cout << "удаление...\n"; }
     int get() { return i*j; }
};
int main()
{
     Samp *p01;
     int i;
     p01 = new Samp(6,5);
     Samp *p02;
     p02 = new Samp[3];
     if(!p01||!p02)
     {
          cout << "ошибка выделения памяти\n";
          return 1;
     }
     for(i=0; i<3; i++)
     {
          p02[i].setIJ(i, i);
          cout     <<     "p02["     <<     i     <<
"]="<<p02[i].get()<<”\n”;
     }
```

```
     cout << p01->get() << "\n";
     delete p01;
     delete [ ] p02;
     return 0;
}
```

Результат:

```
конструктор1
конструктор2
конструктор2
конструктор2
p02[0]=0
p02[1]=1
p02[2]=4
30
удаление...
удаление...
удаление...
удаление...
```

Деструктор вызывается 4 раза: по одному разу на каждый элемент массива и один раз для объекта p01.

Ссылки

Ссылка является скрытым константным указателем и работает как другое имя переменной. Ее можно использовать для передачи аргументов в функцию и возврата значений обратно. При передаче объекта через ссылку в функцию сообщается адрес объекта, а его копия не делается. Это уменьшает вероятность ошибок, связанных с выделением динамической памяти и вызовом деструктора. Аналогично, при возврате ссылки на объект из функции также не делается копия объекта.

При передаче функции объекта в качестве параметра может возникнуть ошибка из-за разрушения деструктором копии объекта, которая должна быть исправлена созданием конструктора копирования. В этой ситуации лучше использовать функцию, возвращающую ссылку на объект. Например:

```
// защищенный двумерный массив
#include <iostream.h>
#include <stdlib.h>
class Array
{
     int isize, jsize;
     int *p;
public:
     Array(int i, int j)
```

```
        {
            p = new int [ i * j ];
            if(!p)
            {
                cout << "Ошибка выделения памяти\n";
                exit(1);
            }
            isize = i;
            jsize = j;
        }
        int &put(int i, int j);
        int get(int i, int j);
    };
    /* возврат ссылки на элемент массива, в который
необходимо выполнить запись */
    int &Array::put(int i, int j)
    {
        if(i<0 || i>=isize || j<0 || j>=jsize)
        {
            cout << "Ошибка, нарушены границы
массива!!!\n";
            exit(1);
        }
        return p[i * jsize + j];// возврат ссылки на p[
i ] [ j ]
    }
    // получение значения из массива
    int Array::get(int i, int j)
    {
        if(i<0 || i>=isize || j<0 || j>=jsize)
        {
            cout << "Ошибка, нарушены границы
массива!!!\n";
            exit(1);
        }
        return p[i * jsize + j]; // возврат символа
    }
    int main()
    {
        Array a(2, 3);
        int i, j;
        for(i=0; i<2; i++)
            for(j=0; j<3; j++)
                a.put(i, j) = i + j;
        for(i=0; i<2; i++)
```

```
            for(j=0; j<3; j++)
                cout << a.get(i, j) << ' ';
        // генерация ошибки нарушения границ массива
        a.put(10, 10);
        return 0;
}
```

Практическое задание №4

Цель работы: Научиться работать с массивами, указателями ссылками.

Постановка задачи: Создать статический или динамический массив, содержащий объекты классов согласно выбранному варианту задания. Реализовать функции ввода и отображения объектов массива.

Варианты заданий

№ варианта	Задание
1	Создать массив фиксированного размера, содержащий объекты типа «Дата». Класс «Дата» должен содержать несколько конструкторов. Инициализировать массив с помощью списка инициализации.
2	Создать массив фиксированного размера, содержащий объекты типа «Время». Класс «Время» должен содержать несколько конструкторов. Инициализировать массив с помощью списка инициализации.
3	Создать динамический массив, содержащий объекты типа «Автомобиль». Класс «Автомобиль» должен содержать несколько конструкторов. Реализовать функцию сортировки массива по какому-либо признаку.
4	Создать динамический массив, содержащий объекты типа «Фигура». Класс «Фигура» должен содержать несколько конструкторов. Реализовать функцию сортировки массива по какому-либо признаку.
5	Создать массив фиксированного размера, содержащий объекты типа «Фигура» и производных от него классов «Точка», «Квадрат» и «Круг». В базовом классе определить методы для отображения и перемещения фигуры. Разработать функцию для отображения и перемещения всех фигур массива.
6	Создать класс, содержащий массив объектов типа «Дата». Реализовать методы доступа к элементам массива с использованием ссылок.
7	Создать класс, содержащий массив объектов типа «Время». Реализовать методы доступа к элементам массива с использованием ссылок.
8	Создать класс, содержащий указатель на динамический массив объектов типа «Фигура». Реализовать методы доступа к элементам массива с использованием ссылок.
9	Создать класс, содержащий указатель на динамический

	массив объектов типа «Транспорт». Реализовать методы доступа к элементам массива с использованием ссылок.
0	Создать класс, содержащий указатель на динамический массив объектов типа «Фигура» и производными от него классами «Точка», «Квадрат» и «Круг». Реализовать методы доступа к элементам массива с использованием ссылок.

Практическое занятие 5. Шаблоны и исключительные ситуации

Шаблоны функций

Шаблоны, которые также называют родовыми или параметризованными типами, позволяют конструировать семейства функций и классов. В отличие от механизма перегрузки, когда для каждого набора формальных параметров устанавливается своя функция, шаблон семейства функций определяется один раз, но при этом оно параметризуется. Параметризовать в шаблоне функции можно тип возвращаемого функцией значения и типы параметров, количество и порядок которых должны быть фиксированы. В определении шаблона употребляется служебное слово template. Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки <>. Каждый формальный параметр шаблона помечен служебным словом class, за которым следует имя параметра.

Шаблон семейства функций состоит из двух частей – заголовка шаблона и определения функции, в котором тип возвращаемого значения и типы параметров обозначаются именами параметров шаблона.

```
template <class Ttype> тип имя_функции(список аргументов)
{/*тело функции*/}
```

Здесь Ttype – фиктивный тип, или список типов, через запятую, который используется при объявлении аргументов, локальных переменных и возвращаемых значений функции. Компилятор заменит этот фиктивный тип на один из реальных и создаст соответственно несколько перегружаемых функций, которые являются ограниченными, поскольку выполняют одни и те же действия. Например:

```
#include <iostream.h>
#include <string.h>
template <class X> int find(X object, X *list, int size)
{
     int i;
     for(i=0; i<size; i++)
         if(object == list[i]) return i;
     return -1;
}
int main()
{
     int a[ ]={1, 2, 3, 4};
     char *c="это проверка";
```

```
        double d[ ]={1.1, 2.2, 3.3};
        cout << find(3, a, 4) << endl;
        cout << find('a', c, strlen(c))<< endl;
        cout << find(0.0, d, 3);
        return 0;
}
```

Компилятор автоматически создает три перегруженные функции find(), соответствующие типам передаваемых аргументов.

Приведем пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template <class type> type abs (type x) { return x > 0 ?
x: -x;}
```

В качестве еще одного примера рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров.

```
template <class T> void swap (T* x, T* y)
{
        T z = *x;
        *x = *y;
        *y = x;
}
```

Здесь параметр T шаблона функций используется не только для спецификации формальных параметров, но и в теле определения функции, где он задает тип вспомогательной переменной z.

Если в программе присутствует приведенный ранее шаблон семейства функций swap() и появится последовательность операторов:

```
long k = 4, d = 8; swap (&k, &d);
```

то компилятор сформирует определение функции:

```
void swap (long* x, long* y)
{
        long z = *x;
        *x = *y;
        *y = z;
}
```

Затем будет выполнено обращение именно к этой функции, и значения переменных k и d поменяются местами.

Если в той же программе присутствуют операторы:

```
double a = 2.44, b = 66.3; swap (&a, &b);
```

то сформируется и выполнится функция

```
void swap (double* x, double* y){

      double x = *x;
      *x = *y;
      *y = x;
}
```

Параметры шаблона

Параметры шаблона являются формальными, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона. Имена параметров шаблона должны быть уникальными во всем его определении. В списке параметров шаблона функций их может быть несколько. Каждый из параметров должен начинаться со служебного слова class. Например, неверен заголовок:

```
template <class type1, type2, type3>
```

Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, например:

```
template <class t, class t, class t>
```

Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Будет ошибочным шаблон:

```
template <class A, class B, class C> B func (A n, C m) {B v; ... }
```

Применение параметра шаблона B в качестве типа возвращаемого функцией значения и для определения объекта v в теле функции недостаточно.

Шаблоны классов

Аналогично шаблонам функций определяется шаблон семейства классов:

```
template <список_параметров_шаблона> класс
```

В определении класса, входящего в шаблон, его имя является не именем отдельного класса, а параметризованным именем семейства классов.

С помощью шаблона класса можно создать класс, реализующий стек, очередь, дерево и т. д. для любых типов данных. Компилятор будет

генерировать правильный тип объекта на основе типа, задаваемого при создании объекта. Объявление шаблона класса имеет вид:

```
template <class Ttype> class имя_класса
{
     // поля и функции класса
}
```

Например:

```
// простой родовой связанный список
#include <iostream.h>
template <class data_t> class List
{
     data_t data;
     List *next;
public:
     List (data_t d);
     void add(List *node)
     {
          node->next = this;
          next=0;
     }
/* новый созданный элемент списка (объект) добавляет
к себе последний элемент, включенный в список */
     List *getnext()
     {
          return next;
     }
     data_t getdata()
     {
          return data;
     }
};
template <class data_t> List<data_t>::List(data_t d)
{
     data=d;
     next=0;
}
int main()
{
     //создается объект с реальным типом данных
     List<char> start('a');
     List <char> *p, *last;
     int i;
```

```
        // создание списка
        last=&start;
        for(i=1; i<26; i++)
        {
            p=new List<char> ('a'+i);
            p->add(last);
            last=p;
        }
        // вывод списка
        p=&start;
        while(p)
        {
            cout << p->getdata();
            p=p->getnext();
        }
        return 0;
}
```

С помощью простого объявления можно создать другой объект, например, для хранения целых:

```
List <int> int start(1);
```

В список можно поместить структуры или другие объекты.
Класс-шаблон может иметь больше одного родового типа данных:

```
template<class T1,class T2> class M {T1 a; T2 b;}
```

В качестве аргумента в общем случае может быть использовано константное выражение, однако выражение, содержащее переменные, использовать в качестве фактического параметра шаблона нельзя.

Исключительные ситуации

Под исключительной ситуацией, или исключением (exception), понимают прерывание нормального потока программного управления в ответ на непредвиденное или аварийное событие. Исключение может порождаться ошибками, такими, как деление числа на нуль или обращением к памяти по недействительному адресу. В качестве ответа на ошибку функция, в которой возникла ошибка, инициирует (возбуждает) исключение оператором throw, за которым следует значение. Это значение может быть константой, переменной или объектом и предназначено для передачи информации обработчику исключения об исключении.

Обработчик исключения начинается ключевым словом catch с объявлением в круглых скобках. Если тип, определенный в этом объявлении,

совпадает с типом значения, данного в операторе throw, управление будет передано в блок, следующий за ключевым словом catch. В случае несовпадения типов, программа осуществляет поиск другого обработчика.

Для процедуры обработки исключения должен быть предусмотрен try-блок, в котором и происходит собственно инициализация исключения. Если исключение не инициализировано в try-блоке, программное управление проигнорирует catch-блок и непосредственно перейдет к первому оператору, находящемуся за catch-блоком.

```
try
{
     /*блок try*/
} catch(type1 arg){/*блок catch*/}
//операторы
```

С блоком try может связываться несколько блоков catch. Выполняется тот блок catch, для которого тип данных соответствует типу возникшей исключительной ситуации. При этом ее значение присваивается аргументу в круглых скобках блока catch. Если ошибка имеет место внутри блока try, она может генерироваться с помощью throw.

```
#include <iostream.h>
int main()
{
     try
     {
          // начало блока try
          cout << "Внутри блока try\n";
          throw 10;    // генерация ошибки 10
          cout << "Это выполнено не будет\n";
     }
     catch (int i)
     {
          // перехват ошибки
          cout << "Перехвачена ошибка номер: "<< i <<
"\n";
     }
     catch (char *s)
     {
          // перехват ошибки
          cout << "Перехвачена ошибка номер: "<<s <<
"\n";
     }
     return 0;
}
```

Будет выведено:

```
Внутри блока try
Перехвачена ошибка: 10
```

Оператор throw генерирует ошибку, после чего управление передано блоку catch. Если заменить тип ошибки int на double, ошибка перехвачена не будет. Для перехвата всех исключительных ситуаций независимо от типа можно воспользоваться многоточием, которое соответствует любому типу данных.

```
catch(…) {/*тело*/}
```

Ключевое слово throw после заголовка функции используется для определения списка инициализированных исключений:

```
void func( ) throw (except1, except2, except3, char*)
{       }
```

Функции, генерирующие исключения, вызываются из блока try. Описанная без оператора throw функция не может корректно для течения программы инициализировать исключение, а та, которая может это сделать, способна вызвать также исключение типа, производного от него. Блок catch, который перехватывает исключение для объектов, может перехватывать и исключения производных типов.

Если в функции инициализируется исключение, тип которого не совпадает с типом ни одного обработчика, то вызывается функция unexpected(), которая в свою очередь, вызывает terminate(), а terminate() – функцию abort() для завершения программы.

Практическое задание №5

Цель работы: Научиться работать с шаблонами классов и функций. Отлавливать и обрабатывать исключительные ситуации.

Постановка задачи: Написать программу с использованием шаблонов или обработчиков исключительных ситуаций согласно выбранному варианту задания.

Варианты заданий

№ варианта	Задание
1	Создать шаблон класса «Однонаправленный список». Реализовать списки объектов класса «Дата».
2	Создать шаблон класса «Стек». Реализовать стек из объектов класса «Время».
3	Создать шаблон класса «Массив» с параметром – количество элементов массива. Реализовать массив из объектов класса «Транспорт».
4	Создать шаблон класса «Двумерный массив» с параметрами – размерность массива. Реализовать массив из объектов класса «Фигура».
5	Создать шаблон функции для сортировки элементов массива. Выполнить сортировку массива объектов класса «Дата»
6	Создать шаблон функции для сортировки элементов массива. Выполнить сортировку массива объектов класса «Время»
7	Создать шаблон функции для сортировки элементов динамического массива. В качестве параметра шаблона указать размер массива .Выполнить сортировку массива объектов класса «Транспорт».
8	Создать класс «Время», содержащий метод для ввода времени с клавиатуры. При вводе некорректного времени, должна генерироваться исключительная ситуация. Реализовать обработчик исключительной ситуации.
9	Создать класс «Дата», содержащий метод для ввода даты с клавиатуры. При вводе некорректной даты, должна генерироваться исключительная ситуация. Реализовать обработчик исключительной ситуации.
0	Создать класс «Массив» для хранения объектов класса

	«Транспорт». Методы доступа к элементам массива должны генерировать исключение в случае некорректного индекса элемента массива. Реализовать обработчик исключительной ситуации.

Практическое занятие 6. Потоки и классы ввода/вывода

Стандартные потоки

В C++ ввод и вывод осуществляется через потоки – объекты классов ввода/вывода, которые передают и принимают данные и связываются с физическими устройствами. Следующие потоки определены и автоматически открываются при запуске приложения:

```
   extern istream cin;   // стандартный поток ввода с
клавиатуры
   extern ostream cout;  // стандартный поток вывода на
экран
   extern ostream cerr;  //стандартный    поток    вывода
сообщений
                     //об ошибках
   extern  ostream  clog;//буферизованный  поток  вывода
сообщений
                     //об ошибках */
```

Рассмотрим пример:

```
#include <iostream.h>
int main()
{
     int a,b,c;
     cout << "Hello, world\n";
     cin>>a>>b>>c;
     cout << "a/b+c=" << a*b-c << "\n";
     cout << "a^b|c=" << (a^b|c) << "\n";
     return 0;
}
```

Операция << пишет аргумент, строку "Hello, world" в стандартный поток вывода cout. Ввод производится с помощью операции >> стандартного потока ввода cin. Приоритет операции вставки в поток << достаточно низок, чтобы не использовать скобки для арифметических выражений. Для операций с более низкими приоритетами без скобок не обойтись.

Форматирование

Операция << применяется для неформатированного вывода данных стандартных типов. Операция определяет тип данных и выбирает подходящий формат. То же происходит и с операцией извлечения из потока >>. Помимо

этого существует несколько функций, преобразующих параметр в строку, которая используется для вывода.

```
    char* oct(long, int=0);    //                восьмеричное
представление
    char* dec(long, int=0);    // десятичное представление
    char* hex(long, int=0);    //          шестнадцатеричное
представление
    char* chr(int, int=0);     // символ
    char* str(char*, int=0);   // строка
```

Второй (необязательный) параметр указывает, сколько символьных позиций должно использоваться. Если задано поле нулевой длины, то для вывода потребуется столько позиций, сколько нужно, иначе будет производиться усечение или дополнение. Например:

```
    cout <<"dec("<< x << ")=oct("<< oct(x,6)<<")= hex("
         << hex(x,4)<< ")";
```

Если x==15, то в результате получится:

```
dec(15) = oct( 17) = hex( f);
```

Для преобразования при выводе можно использовать строку в формате:

```
char* form(char* format, список параметров);
```

Аналогично стандартной функции printf() вывода языка С функция form() возвращает строку, получаемую в результате форматирования параметров, стоящих после первого управляющего параметра – строки формата format. Строка формата состоит из обычных символов, которые просто копируются в поток вывода, и спецификаций преобразования, влекущих преобразование и вывод параметра. Каждая спецификация преобразования начинается с символа %. Например:

```
cout<<form(" x= %d ",x);
```

Манипуляторы

Манипуляторы – функции потока, которые можно включать в операции помещения и извлечения в потоки (<<, >>), они бывают следующие:

```
    endl // помещение в выходной поток символа конца
строки '\n'
    ends // помещение в выходной поток символа '\0'
    flush // вызов функции вывода буферизованных данных
```

```
    dec, hex, oct // установка основания системы счисления
    ws            // игнорирование при вводе пробелов
    setbase(int)  // установка основания системы счисления
    resetiosflasg(long)   // сброс флагов форматирования
по маске
    setiosflags(long) //установка флагов форматирования по
маске
    setfill(int)      // установка заполняющего символа
    setprecision(int) //установка точности вывода
                      //вещественных чисел
    setw(int)         //  установка ширины поля ввода-
вывода
```

Пример вызова манипулятора:

```
    cout << 15 << hex << 15 << setbase(8) << 15;
    cout<<hex<<100<<oct<<100<<dec<<100;
```

Ошибки потоков

Каждый поток (istream или ostream) имеет ассоциированное с ним состояние, с помощью проверки которого осуществляется обработка ошибок и нестандартных условий.

Поток может находиться в одном из следующих состояний:

```
    enum stream_state { _good, _eof, _fail, _bad };
```

Если состояние _good или _eof, последняя операция ввода прошла успешно, а если – _good, то следующая операция ввода может пройти успешно, в противном случае она закончится неудачей. Другими словами, применение операции ввода к потоку, который не находится в состоянии _good, является пустой операцией. Состояние потока можно проверять, например, так:

```
    switch (cin.rdstate())
    {
      case _good: // последняя операция над cin прошла
успешно
           break;
      case _eof: // конец файла
           break;
      case _fail: // некоего рода ошибка форматирования
           break;
      case _bad: // возможно, символы cin потеряны
           break;
```

```
}
```

Файловый ввод-вывод с применением потоков C++

В C++ существуют классы потоков ввода-вывода, определенные в соответствующей библиотеке:

```
ios           //базовый потоковый класс
streambuf     //буферизация потоков
istream       //потоки ввода
ostream       //потоки вывода
iostream      //двунаправленные потоки
istrstream        //строковые потоки ввода
ostrstream        //строковые потоки вывода
strstream     //двунаправленные строковые потоки
ifstream      //файловые потоки ввода
ofstream      //файловые потоки вывода
fstream       //двунаправленные файловые потоки
```

Стандартные потоки (istream,ostream,iostream) служат для работы с терминалом. Строковые потоки (istrstream, ostrstream, strstream) – для ввода-вывода из строковых буферов, размещенных в памяти, файловые потоки (ifstream, ofstream, fstream) – для работы с файлами.

Для реализации файлового ввода-вывода небходимо включить заголовочный файл fstream.h, содержащий производные от istream и ostream классы ifstream, ofstream и fstream, и объявить соответствующие объекты. Например:

```
ifstream in;  //ввод
ofstream out; //вывод
fstream io;   //ввод - вывод
```

Файловые потоки можно определить с помощью конструкторов:

```
ofstream obj (filename, mode),
ifstream obj (filename, mode),
```

где mode может иметь следующие значения:

```
   ios::app      //запись в конец существующего файла
   ios::ate      //после открытия файла перейти в его
конец
   ios::binary   //открыть файл в двоичном режиме
                 //(по умолчанию – текстовый)/
   ios::in       //открыть для чтения
```

```
ios::nocreate //сообщать о невозможности открытия,
              //если файл не существует
ios::noreplace    //сообщать о невозможности открытия,
              //если файл существует
ios::out      //открыть для вывода
ios::trunc        //если   файл   существует,   стереть
содержимое
```

При необходимости изменения способа открытия или применения файла можно при создании файлового потока использовать два или более флагов: ios::app|ios::noreplace

Для открытия файла одновременно для чтения и записи используются объекты класса fstream:

```
fstream obj(filename, ios::in|ios::app);
```

После объявления потоков открытие файла, связывающее его с потоком, можно производить не через конструктор, а с помощью метода open():

```
void open (char *filename,int mode,int access)
```

где filename – имя файла, включающее путь; mode – режим открытия файла, access – доступ. Параметр access принимает следующие значения: 0 – файл со свободным доступом, 1 –только для чтения, 8 – архивный файл.

Например:

```
ofstream out;
out.open("test.dat",ios::out,0);
```

Параметры можно задать по умолчанию.

```
out.open("test.dat");//будет то же самое
```

При завершении программы открытые файлы неявно закрываются. Для явного закрытия объектов файловых потоков применяется метод close().

Ввод-вывод в файлы

Для чтения-записи в потоки можно использовать перегружаемые операторы-функции >> и <<. Например:

```
#include <fstream.h>
int main()
{
```

```
    /* создание  файла вывода с помощью конструктора */
         ofstream fout("test");
         if(!fout)
         {
              cout << "Файл открыть невозможно\n";
              return 1;
         }
         fout << "Привет!\n";
         fout << 100 << ' ' << hex << 100 << endl;
         fout.close();
         ifstream fin("test"); // открытие обычного файла
ввода
         if(!fin)
         {
              cout << "Файл открыть невозможно\n";
              return 1;
         }
         char str[80];
         int i;
         fin >> str >> i;
         cout << str << ' ' << i << endl;
         fin.close();
         return 0;
    }
```

Сам оператор << можно перегрузить, как в следующем примере:

```
    //создание   недружественной   функции-вставки   для
объектов Coord
    include <iostream.h>
    class Coord
    {
         public:
         int x,y;//должны быть открытыми
         Coord()
         {
              x=0;
              y=0;
         }
         Coord(int i,int j)
         {
              x=i;
              y=j;
         }
    };
```

```
//функция вставки для объектов класса coord
ostream &operator<<(ostream &stream, Coord ob)
{
     stream <<ob.x<<”,”<<ob.y<<’\n’;
     return stream;
}
int main()
{
     Coord a(1,1), b(10,23);
     cout <<a<<b;
     return 0;
}
```

Оператор ввода (извлечения) также можно перегрузить:

```
istream &operator>>(istream &stream, имя_класса ob)
{
     //тело функций ввода
     return stream;
}
```

При работе с файлами возможно использование методов ввода-вывода одного символа: istream &get(char &ch); и ostream &put(char ch);

Пример использования этих методов и аргументов командной строки:

```
#include <iostream.h>
#include <fstream.h>
int main(int argc, char *argv[ ])
{
     char ch;
     if(argc!=2)
     {
          cout << "Использование:WRITE<имя_файла>\n";
          return 1;
     }
     ofstream out(argv[1]);
     if(!out)
     {
          cout << "Файл открыть невозможно\n";
          return 1;
     }
     cout << "Для остановки введите символ $\n";
     do
     {
          cout << ": ";
```

```
            cin.get(ch);
            out.put(ch);
        } while (ch!='$');
        out.close();
        return 0;
    }
```

Для записи-считывания блоков двоичных данных используются функции, которые считывают-записывают n байт в буфер или из буфера:

```
    istream &read(unsigned char *buf, int n);
    ostream &write(const unsigned char *buf, int n);

    #include <iostream.h>
    #include <fstream.h>
    #include <string.h>
    int main()
    {
        //   не   будет   нежелательных   преобразований
символов
        // при вводе и выводе
        ofstream out("test", ios::binary);
        if(!out)
        {
            cout << "Файл открыть невозможно\n";
            return 1;
        }
        double num = 100.45;
        char str[ ] = "Это проверка";
        out.write((char *) &num, sizeof(double));
        out.write(str, strlen(str));
        out.close();
        return 0;
    }
```

Для позиционирования в файле имеются методы seekg(), seekp(), tellp(). При этом seekg() назначает или возвращает текущую позицию указателя чтения, а seekp() – текущую позицию указателя записи. Обе функции могут иметь один или два аргумента. При вызове с одним аргументом функции перемещают указатель на заданное место, а при вызове с двумя аргументами вычисляется относительная позиция от начала файла (ios::beg) текущей позиции (ios::cur) или от конца файла (ios::end). Текущая позиция определяется методом tellp().

Для объектов файловых потоков контроль состояния производится с помощью методов, манипулирующих флагами ошибок:

bad() – возвращает ненулевое значение, если обнаружена ошибка;
clear() – сбрасывает сообщения об ошибках;
eof() – возвращает ненулевое значение, если обнаружен конец файла;
fail() – возвращает ненулевое значение, если операция завершилась неудачно;
good() – возвращает ненулевое значение, если флаги ошибок не выставлены;
rdstate() – возвращает текущее состояние флагов ошибок.

Если флаги показывают наличие ошибки, все попытки поместить в поток новые объекты будут игнорироваться.

Практическое задание №6

Цель работы: Научиться использовать потоки и классы ввода-вывода стандартной библиотеки языка C++.

Постановка задачи: Написать программу с использованием потоков и классов согласно выбранному варианту задания.

Варианты заданий

№ варианта	Задание
1	Создать класс «Массив» для хранения строк. Реализовать методы для вывода массива строк в текстовом виде и в виде шестнадцатеричного дампа.
2	Создать класс «Стек» для хранения строк. Реализовать методы для вывода содержимого стека строк в текстовом виде и в виде шестнадцатеричного дампа.
3	Создать класс «Однонаправленный список» для хранения строк. Реализовать методы для вывода массива строк в текстовом виде и в виде последовательности десятичных кодов символов.
4	Создать класс «Стек» для хранения комплексных чисел. Реализовать методы чтения содержимого стека из файла и записи содержимого стека в файл.
5	Создать класс «Массив» для хранения векторов. Реализовать методы чтения массива векторов из файла и записи массива векторов в файл.
6	Создать класс «Однонаправленный список» для хранения объектов класса «Время». Реализовать методы чтения объектов списка из файла и записи объектов списка в файл.
7	Создать класс «Стек» для хранения объектов класса «Дата». Реализовать методы чтения содержимого стека из файла и записи содержимого стека в файл.
8	Создать класс «Массив» для хранения объектов класса «Транспорт». Реализовать методы чтения объектов массива из файла и записи объектов массива в файл.
9	Создать массив фиксированного размера для хранения объектов класса «Фигура» и производных от него: «Точка», «Квадрат», «Круг». Написать функцию для записи массива объектов в файл и чтения массива

	объектов их файла.
0	Создать динамический массив для хранения объектов класса «Транспорт» и производных от него: «Автомобиль», «Самолет», «Поезд». Написать функцию для записи массива объектов в файл и чтения массива объектов их файла.

Практическое занятие 7. Статические и константные члены, локальные классы

Статические члены

Класс – это тип, а не объект данных, и в каждом объекте класса имеется своя собственная копия данных, членов этого класса. Иногда необходимо, чтобы некоторые данные-члены класса были одинаковыми (разделяемыми) для всех экземпляров класса. Предпочтительно, чтобы такие разделяемые данные были описаны как часть класса. Возможно применение статических методов класса, которые можно вызывать не через объект (даже тогда, когда объект существует), а через имя класса. Для объявления членов класса статическим применяется модификатор static. Статические данные-члены класса должны быть объявлены в классе и определены за пределами класса (внутренние определения запрещены). Обычные спецификаторы доступа действуют и для статических членов класса. Память, занимаемая статическими данными класса, не учитывается при определении размера объекта с помощью операции sizeof.

```
class My
{
     static int count;
public:
     static int getCount(){return count;}
};
int My::count = 0;
```

Так как статические элементы-функции не ассоциируются с отдельными представителями, то при вызове им не передается указатель this. Из этого следует, что:

1. статическая функция член класса может обращаться только к статическим данным класса и вызывать только другие статические функции текущего класса;
2. статическая функция не может быть объявлена как virtual или const.

Использование статических членов класса может заметно снизить потребность в глобальных переменных, применение которых нежелательно в принципе.

Константные члены и объекты

Можно создавать объекты класса с модификатором const, информирующим компилятор о том, что содержимое объекта не должно изменяться после инициализации. Чтобы предотвратить изменение значений элементов константного объекта, компилятор сообщает об ошибке, если объект

используется с неконстантной функцией-элементом. Возможно использование константных указателей и указателей на константный объект.

```
    My *p01;// константный указатель, который нельзя изменять
    p01= new My;//константный объект
    My* const p03=p01;
    const My ob;
    const My *p04=&ob;
    /* указатель на константный объект, т.е. объект нельзя изменить, используя данный указатель, но можно изменять сам указатель p04*/
    const My* const p01=&ob;
    /* указатель-константа на константный объект: нельзя изменять и вызывать неконстантные функции*/
```

Константная функция-элемент в свою очередь:

1. объявляется с ключевым словом const, которое следует за списком параметров;
2. не может изменять значение полей класса;
3. не может вызывать неконстантные функции-члены класса;
4. может вызываться как для константных, так и неконстантных объектов класса.

Следующий пример демонстрирует константные функции и объекты:

```
    class Point1
    {
         int x;
    public:
         Point1(int x){Point1::x=x;}
         void setX(int x){Point1::x=x;}
         void getX(int &x1)const;
    };
    void Point1::getX(int &x1) const
    {
         x1=x;
    }
    int main(int argc, char* argv[])
    {
         Point1 ob1(5);
         const Point1 ob2(10);
         int x2;
         ob1.getX(x2);
         //ob2.setX(x2);//ошибка:  вызов  неконстантной функции
```

```
                    // неконстантным объектом
        ob2.getX(x2);
        return 0;
    }
```

Однако следует отметить, что константный метод может изменять данные-члены класса, которые описаны с модификатором mutable, предшествующим типу данных:

```
mutable int i;
```

Локальные классы

Если класс объявляется вне любого блока, то он называется глобальным. В то же время класс может объявляться внутри функции, называясь при этом внутренним. Область видимости внутреннего класса ограничивается функцией, в которой он определен. Если класс объявляется внутри другого класса, то он является вложенным. Его область видимости – класс, в котором он определен. Внутренние и вложенные классы называют локальными.

Пример описания вложенного класса:

```
class Student
{
      int id;
      class Exam
      {
          int idExam;
          char name[80];
      public:
          Exam (int idExam, char* s)
          {
              Exam::idExam=idExam;//определение
видимости
              strcpy(name, s);
          }
      };
      Exam* first;
public:
      void add(int n, char *s)
      {
          first = new Exam(n, s);
      }
};
```

Внутри локальных классов можно использовать типы, статические и внешние (extern) переменные, внешние функции и элементы перечислений из области, где локальный класс описан. В то же время нельзя использовать автоматические переменные из указанной области. В локальных классах можно определить статические методы. Во вложенных классах можно также использовать статические переменные. Внутренний класс не может иметь статических переменных. Методы локального класса определяются только внутри класса.

Если один класс вложен в другой класс, то это не дает каких-либо особых прав доступа к элементам друг друга. Обращение может выполняться по общим правилам.

```
class OuterClass
{
     class InnerClass
     {
          //во вложенном классе можно использовать
          //статические переменные
          static double d;
     };//конец объявления вложенного класса
}; // конец объявления внешнего класса
     /*определение статической переменной */
     double OuterClass::InnerClass::d=5.32;
     void extf()
     {
          //Внешняя функция
          class InnerClass1
          {
          static double d1;    //ошибка:         нельзя
определить
                              //переменную      d1      за
пределами
                                   //функции extf();
          }; //конец объявления внутреннего класса
};
int main()
{
     OuterClass oc;
     extf();
}
```

Если только вложенный класс не является очень простым, то в таком описании трудно разобраться. Рекомендуется не использовать сложные описания локальных классов и не использовать в иерархии вложенности более одной ступени. Кроме того, вложение классов – это не более чем соглашение о

записи, поскольку вложенный класс не является скрытым в области видимости лексически охватывающего класса.

```
class Student
{
     int id;
     class Exam
     {
          int idExam;
          char name[80];
     public:
          Exam (int idExam, char* s);
     };
     Exam* first;
public:
     void add(int n, char *s)
     {
          first = new Exam(n, s);
     }
};
Student::Exam::Exam (int idExam, char* s)
{
     Exam::idExam=idExam;
     strcpy(name, s);
}
```

Большую часть нетривиальных классов лучше описывать раздельно, и при этом существует возможность обеспечить доступ к закрытым данным-членам некоторого класса сразу для всех методов другого класса. Такой класс объявляется дружественным для первого класса.

```
class Exam
{
     friend class Student;//объявление дружественного
класса
     int idExam;
     char name[80];
public:
     Exam (int idExam, char* s)
     {
          Exam::idExam=idExam;
          strcpy(name, s); }
     };
class Student
{
```

```
     int id;
     Exam* first;
public:
     void add(int n, char *s)
     {
          first = new Exam(n, s);
     }
};
```

Практическое задание №7

Цель работы: Научиться работать со статическими и константными членами классов. Научиться объявлять локальные классы.

Постановка задачи: Написать программу с использованием статических и константных членов класса, а также локальных классов. Создать класс согласно выбранному варианту задания. Реализовать методы чтения и отображения данных.

Варианты заданий

№ варианта	Задание
1	Создать класс «Однонаправленный список» для хранения объектов класса «Время». Класс «Время» должен содержать статический член данных для подсчета количества экземпляров объектов данного класса.
2	Создать класс «Стек» для хранения объектов класса «Дата». Класс «Дата» должен содержать статический член данных для подсчета количества экземпляров объектов данного класса.
3	Создать класс «Стек» для хранения объектов класса «Транспорт». Класс «Транспорт» должен содержать статический член данных для подсчета количества объектов данного класса.
4	Создать класс «Время», содержащий только один приватный конструктор. Реализовать статический метод для создания объектов класса.
5	Создать класс «Дата», содержащий только один приватный конструктор. Реализовать статический метод для создания объектов класса.
6	Создать класс «Транспорт», содержащий только один приватный конструктор. Реализовать статический метод для создания объектов класса.
7	Создать класс «Дата и Время», содержащий вложенные классы «Дата» и «Время».
8	Создать класс «Транспортные средства», содержащий вложенные классы «Автомобиль», «Самолет», «Поезд».

9	Создать класс «Фигуры», содержащий вложенные классы «Точка», «Квадрат», «Круг».
0	Создать класс «Компьютер», содержащие вложенные классы, описывающие компоненты компьютера.

Практическое занятие №8. Работа с файлами.

Цель: овладеть возможностью считывания/записи данных из/в файл.

Ход работы:

1. Рассмотреть описанные в теоретических сведениях примеры.
2. Реализовать возможность записи/чтения экземпляра класса, созданного самостоятельно.

Теоретические сведения.

Потоки: байтовые, символьные, двоичные

Большинство устройств, предназначенных для выполнения операций ввода–вывода, являются байт–ориентированными. Этим и объясняется тот факт, что на самом низком уровне все операции ввода–вывода манипулируют с байтами в рамках **байтовых потоков**.

С другой стороны, значительный объём работ, для которых, собственно и используется вычислительная техника, предполагает работу с символами, а не с байтами (заполнение экранной формы, вывод информации в наглядном и легко читаемом виде, текстовые редакторы).

Символьно–ориентированные потоки, предназначенные для манипулирования с символами, а не с байтами, являются потоками ввода–вывода более высокого уровня. В рамках Framework.NET определены соответствующие классы, которые при реализации операций ввода–вывода обеспечивают автоматическое преобразование данных типа byte в данные типа char и обратно.

В дополнение к байтовым и символьным потокам в C# определены два класса, реализующих механизмы считывания и записи информации непосредственно в двоичном представлении (потоки BynaryReader и BynaryWriter).

Общая характеристика классов потоков

Основные особенности и правила работы с устройствами ввода–вывода в современных языках высокого уровня описываются в рамках классов потоков. Для языков платформы .NET местом описания самых общих свойств потоков является **класс System.IO.Stream**.

Назначение этого класса заключается в объявлении общего стандартного набора операций (стандартного интерфейса), обеспечивающих работу с устройствами ввода–вывода, независимо от их конкретной реализации источников и получателей информации.

В рамках Framework.NET, независимо от характеристик того или иного устройства ввода–вывода, **программист ВСЕГДА может узнать**:

можно ли читать из потока – bool CanRead (если можно – значение должно быть установлено в true),

можно ли писать в поток – bool CanWrite (если можно – значение должно быть установлено в true),

можно ли задать в потоке текущую позицию – bool CanSeek (если последовательность, в которой производится чтение–запись не является жёстко детерминированной и возможно позиционирование в потоке – значение должно быть установлено в true),

позицию текущего элемента потока – long Position (возможность позиционирования в потоке предполагает возможность программного изменения значения этого свойства),

общее количество символов потока (длину потока) – long Length.

В соответствии с общими принципами реализации операций ввода–вывода, **для потока** предусмотрен **набор методов, позволяющих реализовать**:

Метод потока	Назначение
int ReadByte()	чтение байта из потока с возвращением целочисленного представления СЛЕДУЮЩЕГО ДОСТУПНОГО байта в потоке ввода
int Read(byte[] buff, int index, int count)	чтение определённого (count) количества байтов из потока и размещение их в массиве buff, начиная с элемента buff[index], с возвращением количества успешно прочитанных байтов
Write(byte[] buff, int index, int count)	запись в поток одного байта
int void WriteByte(byte b)	запись в поток определённого (count) количества байтов из массива buff, начиная с элемента buff[index], с возвращением количества успешно записанных байтов
long Seek (long index, SeekOrigin origin)	позиционирование в потоке –(позиция текущего байта в потоке задаётся значением смещения index относительно позиции origin)
void Flush()	для буферизованных потоков принципиальна операция флэширования (записи содержимого буфера потока на физическое устройство)
void Close()	закрытие потока

Множество классов потоков ввода–вывода в Framework.NET основывается (наследует свойства и интерфейсы) на **абстрактном классе Stream**. При этом классы конкретных потоков обеспечивают собственную реализацию интерфейсов этого абстрактного класса.

Наследниками класса Stream являются, в частности, три класса байтовых потоков:

BufferedStream – обеспечивает буферизацию байтового потока. Как правило, буферизованные потоки являются более производительными по сравнению с небуферизованными,

FileStream – байтовый поток, обеспечивающий файловые операции ввода–вывода,

MemoryStream – байтовый поток, использующий в качестве источника и хранилища информации оперативную память.

Манипуляции с потоками предполагают НАПРАВЛЕННОСТЬ производимых действий. Информацию из потока можно ПРОЧИТАТЬ, а можно её в поток ЗАПИСАТЬ. Как чтение, так и запись, предполагают реализацию определённых механизмов байтового обмена с устройствами ввода–вывода.

Свойства и методы, объявляемые в соответствующих классах, **определяют специфику потоков**, используемых для чтения и записи:

TextReader,

TextWriter.

Эти классы являются абстрактными. Это означает, что они не "привязаны" ни к какому конкретному потоку. Они лишь определяют интерфейс (набор методов), который позволяет организовать чтение и запись информации для любого потока.

Класс Filestream

В C# предусмотрены классы, которые позволяют считывать содержимое файлов и записывать в них информацию. Конечно же, дисковые файлы — самый распространенный тип файлов. На уровне операционной системы все файлы обрабатываются на побайтовой основе.

Чтобы создать байтовый поток с привязкой к файлу, используйте класс FileStream. Класс **FileStream** — производный от Stream и потому обладает функциональными возможностями базового класса. Помните, что потоковые классы, включая FileStream, определены в пространстве имен **System.IO**.

Следовательно, при их использовании в начало программы вы должны включить следующую инструкцию:

using System.IO;

Открытие и закрытие файла.

Для начала необходимо создать объект класса **FileStream**, выбрав наиболее подходящий конструктор. Самый распространенный:

```
FileStream(string filename, FileMode mode)
```

Например, `FileStream fin = new FileStream("test.dat", FileMode.Open);`

Значения FileMode enumeration, описывающие, каким образом операционная система должна открывать файл:

Имя члена	Описание
Append	Открывается существующий файл, и выполняется поиск конца файла, или же создается новый файл. FileMode.Append можно использовать только вместе с FileAccess.Write. Любая попытка чтения заканчивается неудачей и создает исключение

	ArgumentException.
Create	Описывает, что операционная система должна создавать новый файл. Если файл уже существует, он будет переписан. Это требует FileIOPermissionAccess.Write и FileIOPermissionAccess.Append. System.IO.FileMode.Create эквивалентно следующему запросу: если файл не существует, использовать CreateNew; в противном случае использовать Truncate.
CreateNew	Описывает, что операционная система должна создать новый файл. Это требует FileIOPermissionAccess.Write. Если файл уже существует, создается исключение IOException.
Open	Описывает, что операционная система должна открыть существующий файл. Возможность открыть данный файл зависит от значения, задаваемого FileAccess. Если данный файл не существует, создается исключение System.IO.FileNotFoundException.
OpenOrCreate	Указывает, что операционная система должна открыть файл, если он существует, в противном случае должен быть создан новый файл. Если файл открыт с помощью FileAccess.Read, требуется FileIOPermissionAccess.Read. Если файл имеет доступ FileAccess.ReadWrite и данный файл существует, требуется FileIOPermissionAccess.Write. Если файл имеет доступ FileAccess.ReadWrite и файл не существует, в дополнение к Read и Write требуется FileIOPermissionAccess.Append .
Truncate	Описывает, что операционная система должна открыть существующий файл. После открытия должно быть выполнено усечение файла таким образом, чтобы его размер стал равен нулю. Это требует FileIOPermissionAccess.Write. Попытка чтения из файла, открытого с помощью Truncate, вызывает исключение.

Если необходимо ограничить доступ только чтением или только записью, используйте следующий конструктор:

FileStream(string filename, FileMode mode,FileAccess how)

Где FileAccess enumerations определяет доступ к файлу:

Члены перечисления	Описание
Read	Read access to the file. Data can be read from the file. Combine with Write for read/write access.
ReadWrite	Read and write access to the file. Data can be written to and read from the file.
Write	Write access to the file. Data can be written to the file. Combine with Read for read/write access.

Например,

```
FileStream fin = new FileStream("test.dat",FileMode.Open,FileAccess.Read)
```

По завершении работы с файлом его необходимо закрыть. Для этого достаточно вызвать метод Close(). При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для других файлов. Метод Close() может генерировать исключение типа lOException.

```
fin.Close();
```

В **классе FileStream** определены два метода, которые считывают/записывают байты из файла:

ReadByte() / WriteByte() - чтобы прочитать/записать из файла один байт

Read () / Write() - чтобы считать/записать блок байтов.

При выполнении операции вывода в файл выводимые данные зачастую не записываются немедленно на реальное физическое устройство, а буферизируются операционной системой до тех пор, пока не накопится порция данных достаточного размера, чтобы ее можно было всю сразу переписать на диск. Такой способ выполнения записи данных на диск повышает эффективность системы. Например, дисковые файлы организованы по секторам, которые могут иметь размер от 128 байт. Данные, предназначенные для вывода, обычно буферизируются до тех пор, пока не накопится такой их объем, который позволяет заполнить сразу весь сектор.

Но если вы хотите записать данные на физическое устройство вне зависимости от того, полон буфер или нет, вызовите следующий метод Flush ():

```
void Flush()
```

```
// Запись данных в файл.
   class WriteToFile
   {
      public static void Main(string[] args)
      {
         FileStream fout;
         // Открываем выходной файл
         try
         {
            fout = new FileStream("test.txt", FileMode.Create);
         }
         catch (IOException exc)
         {
            Console.WriteLine(exc.Message + "Ошибка при открытии выходного файла.");
            return;
         }
```

```
            // Записываем в файл алфавит,
            try
            {
               for (char c = 'A'; c <= 'Я'; c++) fout.WriteByte((byte)c);
            }
            catch (IOException exc)
            {
               Console.WriteLine(exc.Message + "Ошибка при записи в
файл.");
               fout.Close();
            }

         }
      }
```

Эта программа сначала открывает для вывода файл с именем test.txt . Затем в этот файл записывается алфавит английского языка, после чего файл закрывается.

Обратите внимание на то, как обрабатываются возможные ошибки с помощью блоков try/catch.

Копирование файла.

Одно из достоинств байтового ввода-вывода с использованием класса FileStream заключается в том, что этот класс можно использовать для всех типов файлов, а не только текстовых.

```
   public static void Main(string[] args)
      {
         int i;
         FileStream fin = new FileStream(args[0], FileMode.Open);
         FileStream fout = new FileStream(args[1], FileMode.Create);
         // Копируем файл,
         try
         {
            do
            {
               i = fin.ReadByte();
               if (i != -1) fout.WriteByte((byte)i);
            } while (i != -1);
         }
         catch (IOException exc)
         {
            Console.WriteLine(exc.Message + "Ошибка при чтении файла. ");
            fin.Close();
            fout.Close();
         }
```

Файловый ввод–вывод с ориентацией на символы.

Следующие методы определяют базовые механизмы символьного ввода–вывода.

Для класса TextReader:

Методы класса **TextReader**	Назначение
int Peek()	Reads the next character without changing the state of the reader or the character source. Returns the next available character without actually reading it from the input stream
int Read()	Перегруженные. Несколько одноименных функций с одним и тем же именем. Читает значения из входного потока. Вариант int Read() предполагает чтение из потока одного символа с возвращением его целочисленного эквивалента или –1 при достижении конца потока. Вариант int Read (char[] buff, int index, int count) и его полный аналог int ReadBlock(char[] buff, int index, int count) обеспечивает прочтение a maximum of count characters из текущего потока и записывает данные в buffer, начиная at index
string ReadLine()	Читает строку символов из текущего потока. Возвращается ссылка на объект типа string
string ReadToEnd()	Читает все символы, начиная с текущей позициисимвольного потока, определяемого объектом класса TextReader и возвращает результат как ссылка на объект типа string
void Close()	Закрывает поток ввода

Для класса TextWriter:

Методы класса **TextWriter**	Назначение
void Write()	множество перегруженных вариантов функции со значениями параметров, позволяющих записывать символьное представление значений базовых типов (смотреть список базовых типов) и массивов значений (в том числе и массивов строк)
void Flush()	Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream. Очистка буфера вывода с предварительным выводом в поток вывода (носитель данных) содержимого буфера
void Close()	Закрывает поток вывода.

Эти классы являются базовыми для классов:

StreamReader – содержит свойства и методы, обеспечивающие считывание СИМВОЛОВ из байтового потока,

StreamWriter – содержит свойства и методы, обеспечивающие запись СИМВОЛОВ в байтовый поток.

Интересно заметить, что у всех ранее перечисленных классов имеются методы, обеспечивающие закрытие потоков и не определены методы обеспечивающие открытие соответствующего потока. Потоки открываются в момент создания объекта–представителя соответствующего класса. Наличие функции, обеспечивающей явное закрытие потока принципиально. Оно связано с особенностями выполнения управляемых модулей в Framework.NET. Время начала работы сборщика мусора заранее неизвестно.

Пример использования StreamWriter.

Рассмотрим простую утилиту, которая считывает строки текста, вводимые с клавиатуры, и записывает их в файл test.txt. Текст считывается до тех пор, пока пользователь не введет слово "стоп". Здесь используется объект класса FileStream, помещенный в оболочку класса StreamWriter для вывода данных в файл.

```
public static void Main(string[] args)
        {
           string str;
           FileStream fout = new FileStream("test.txt", FileMode.Create);
           StreamWriter fstr_out = new StreamWriter(fout);
           Console.WriteLine("Введите текст 'стоп' для завершения).");
           do
           {
              Console.Write(": ");
              str = Console.ReadLine();
              if (str != "стоп")
              {
                 str = str + "\r\n"; // Добавляем символ новой строки,
                 try
                 {
                    fstr_out.Write(str);
                 }
                 catch (IOException exc)
                 {
                    Console.WriteLine(exc.Message + "Ошибка при работе с
файлом.");
                    return;
                 }
              }
           }  while (str != "стоп");
            fstr_out.Close() ;
           }
```

Пример использования StreamReader.

Следующая программа считывает текстовый файл test.txt и отображает его содержимое на экране.

```
public static void Main(string[] args)
      {
         string s;
         FileStream fin = new FileStream("test.txt", FileMode.Open);
         StreamReader fstr_in = new StreamReader(fin);
         // Считываем файл построчно
         while ((s = fstr_in.ReadLine()) != null)
         {
            Console.WriteLine(s);
         }
         fstr_in.Close();

      }
```

Пример перенаправления потоков.

Такие стандартные потоки, как Console.In, можно перенаправлять. Безусловно, чаще всего они перенаправляются в какой-нибудь файл. При перенаправлении стандартного потока входные и/или выходные данные автоматически направляются в новый поток. При этом устройства, действующие по умолчанию, игнорируются. Благодаря перенаправлению стандартных потоков программа может считывать команды из дискового файла, создавать системные журналы или даже считывать входные данные с сетевых устройств.

Перенаправить стандартный поток можно двумя способами. Во-первых, при вы- полнении программы из командной строки можно использовать операторы "<" и ">".

В случае программного перенаправления потоков используются:

static void Setln(TextReader *input)*

static void SetOut(TextWriter *output)*

static void SetError(TextWriter *output)*

Пример, который перенаправляет выходной поток в файл:

```
// Перенаправление потока Console.Out.
         StreamWriter log_out;
         try
         {
            log_out = new StreamWriter("logfile.txt");
         }
         catch (IOException exc)
         {
            Console.WriteLine(exc.Message + "Не удается открыть файл.");
            return;
         }
         // Направляем стандартный выходной поток в системный журнал.
```

```
            Console.SetOut(log_out);
            Console.WriteLine("Это начало системного журнала.");
            for(int i=0; i<10; i++) Console.WriteLine(i);
            Console.WriteLine("Это конец системного журнала.");
                log_out.Close();
```

Содержимым файла logfile.txt будет:

```
Это начало системного журнала.
0
1
2
3
4
5
6
7
8
9
Это конец системного журнала.
```

И в файл можно записать информацию, используя привычные классы и методы!

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace OutFile
{
class Program
{

static void Main(string[] args)
{
string buff;

FileStream outFstr, inFstr; // Ссылки на файловые потоки.

// Ссылка на выходной поток. Свойства и методы,
// которые обеспечивают запись в...
StreamWriter swr;

// Ссылка на входной поток. Свойства и методы,
// которые обеспечивают чтение из...
```

```
StreamReader sr;

// Класс Console - Средство управления ПРЕДОПРЕДЕЛЁННЫМ потоком.
// Сохранили стандартный выходной поток,
// связанный с окошком консольного приложения.
TextWriter twrConsole = Console.Out;

// Сохранили стандартный входной поток, связанный с буфером клавиатуры.
TextReader trConsole = Console.In;

inFstr = new FileStream
        (@"F:\Users\Work\readme.txt", FileMode.Open, FileAccess.Read);
sr = new StreamReader(inFstr); // Входной поток, связанный с файлом.

outFstr = new FileStream
        (@"F:\Users\Work\txt.txt", FileMode.Create, FileAccess.Write);
swr = new StreamWriter(outFstr); // Выходной поток, связанный с файлом.

// А вот мы перенастроили предопределённый входной поток.
// Он теперь связан не с буфером клавиатуры, а с файлом, открытым для чтения.
Console.SetIn(sr);
Console.SetOut(swr);

while (true)
{
// Но поинтересоваться в предопределённом потоке относительно
// конца файла невозможно.
// Такого для предопределённых потоков просто не предусмотрено.
if (sr.EndOfStream) break;

// А вот читать - можно.
buff = Console.ReadLine();
Console.WriteLine(buff);

}


Console.SetOut(twrConsole);
Console.WriteLine("12345");
Console.SetOut(swr);
Console.WriteLine("12345");
Console.SetOut(twrConsole);
```

```
Console.WriteLine("67890");
Console.SetOut(swr);
Console.WriteLine("67890");

sr.Close();
inFstr.Close();
swr.Close();
outFstr.Close();
}
}
}
```

ЛИТЕРАТУРА

Перечень основной литературы:

1. Иванова Г.С. Объектно-ориентированное программирование [Электронный ресурс]: учебник/ Иванова Г.С., Ничушкина Т.Н. — Электрон. текстовые данные. — Москва: Московский государственный технический университет имени Н.Э. Баумана, 2014. — 456 с. — Режим доступа: http://www.iprbookshop.ru/94030.html. — ЭБС «IPRbooks».
2. Маляров А.Н. Объектно-ориентированное программирование [Электронный ресурс]: учебник для технических вузов/ Маляров А.Н.— Электрон. текстовые данные. — Самара: Самарский государственный технический университет, ЭБС АСВ, 2017.— 332 с.— Режим доступа: http://www.iprbookshop.ru/91772.html.— ЭБС «IPRbooks».
3. Мурадханов С.Э. Информатика и программирование: объектно-ориентированное программирование (на основе языка C#) [Электронный ресурс]: учебник/ Мурадханов С.Э., Широков А.И. — Электрон. текстовые данные. — Москва: Издательский Дом МИСиС, 2015. — 309 с. — Режим доступа: http://www.iprbookshop.ru/98855.html.— ЭБС «IPRbooks».

Перечень дополнительной литературы:

1. Зыков С.В. Введение в теорию программирования. Объектно-ориентированный подход [Электронный ресурс]: учебное пособие/ Зыков С.В.— Электрон. текстовые данные.— Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021.— 187 с.— Режим доступа: http://www.iprbookshop.ru/102007.html.— ЭБС «IPRbooks».
2. Николаев Е.И. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие/ Николаев Е.И. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2015. — 225 с. — Режим доступа: http://www.iprbookshop.ru/62967.html. — ЭБС «IPRbooks».
3. Сорокин А.А. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие. Курс лекций/ Сорокин А.А. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2014. — 174 с. — Режим доступа: http://www.iprbookshop.ru/63110.html. — ЭБС «IPRbooks».