

Лекция 10. Классы в C++. Объектно-ориентированное программирование.

10.1. Общие сведения о классах

Класс - составной тип данных, элементами которого являются функции и переменные. В основу понятия класс положен тот факт, что «над объектами можно совершать различные операции». Свойства объектов описываются с помощью полей классов, а действия над объектами описываются с помощью функций, которые называются методами класса. Класс имеет *имя*, состоит из полей, называемых *членами класса* и функций - *методов класса*.

Описание класса имеет следующий формат:

```
class name // name – имя класса
{
private:
    // Описание закрытых членов и методов класса
protected:
    // Описание защищенных членов и методов класса
public:
    // Описание открытых членов и методов класса
}
```

10.2. Открытые и закрытые члены класса

В отличие от полей структуры доступных всегда, в классах могут быть члены и методы различного уровня доступа:

- *открытые* `public` (публичные), вызов открытых членов и методов класса осуществляется с помощью оператора `.` ("точка");
- *закрытые* `private` (приватные), доступ к которым возможен только с помощью открытых методов.
- *защищенные* методы (`protected`).

После описания класса необходимо описать переменную типа `class`.

Например,

```
name_class name;
```

здесь `name_class` – имя класса, `name` – имя переменной.

В дальнейшем переменную типа `class` будем называть «*объект*» или «*экземпляр класса*». Объявление переменной типа `class` (в нашем примере переменная `name` типа `name_class`) называется *созданием (инициализацией) объекта (экземпляра класса)*.

После описания переменной можно обращаться к членам и методам класса. Обращение к членам и методам класса осуществляется аналогично обращению к полям структуры с помощью оператора «`.`» (точка).

```

name.p1;           //Обращение к полю p1
                  //экземпляра класса name.
name.f1(par1,par2,...parn); //Обращение к методу f1
                  //экземпляра класса name,
//par1, par2, ..., parn - список формальных
//параметров функции f1.

```

Члены класса доступны из любого метода класса и их не надо передавать в качестве параметров функций-методов.

ЗАДАЧА 10.1. Рассмотрим класс `complex` для работы с комплексными числами¹.

В классе `complex` будут

члены класса:

- `double x` – действительная часть комплексного числа;
- `double y` – мнимая часть комплексного числа.

методы класса:

- `double modul()` – функция вычисления модуля комплексного числа;
- `double argument()` – функция вычисления аргумента комплексного числа;
- `void show_complex()` – функция выводит комплексное число на экран.

Ниже приведен текст класса и функция `main`, демонстрирующая работу с классом.

```

#include <iostream>
#include <string>
#include <math.h>
#define PI 3.14159
using namespace std;
class complex //Определяем класс complex
{
public:
double x;    //Действительная часть комплексного числа.
double y;    //Мнимая часть комплексного числа.
    //Метод класса complex - функция modul,
    //для вычисления модуля комплексного числа.
double modul()
{
return pow(x*x+y*y,0.5);
}
    //Метод класса complex - функция argument,
    //для вычисления аргумента комплексного числа.
double argument()

```

¹Освежить знания о комплексных числах, можно на странице http://kvant.mccme.ru/1982/03/kompleksnye_chisla.htm.

```

{
return atan2(y,x)*180/PI;
}
//Метод класса complex - функция show_complex,
//для вывода комплексного числа.
void show_complex()
{
if (y>=0)
//Вывод комплексного числа с положительной
//мнимой частью.
cout<<x<<"+"<<y<<"i"<<endl;
else
//Вывод комплексного числа с отрицательной
//мнимой частью.
cout<<x<<y<<"i"<<endl;
}
};
int main()
{
//Определяем переменную chislo типа complex.
complex chislo;
//Определяем действительную часть комплексного числа.
chislo.x=3.5;
//Определяем мнимую часть комплексного числа.
chislo.y=-1.432;
//Вывод комплексного числа, chislo.show_complex() -
//обращение к методу класса.
chislo.show_complex();
//Вывод модуля комплексного числа, chislo.modul() -
//обращение к методу класса.
cout<<"Modul' chisla="<<chislo.modul();
//Вывод аргумента комплексного числа,
//chislo.argument() - обращение к методу класса.
cout<<endl<<"Argument chisla="<<chislo.argument()<<endl;
return 1;
}

```

Результат работы программы:

3.5-1.432i

Modul chisla=3.78162

Argument chisla=-22.2516

Использование открытых членов и методов позволяет получить полный доступ к элементам класса, однако это не всегда хорошо. Если все члены класса объявить открытыми, то при непосредственном обращении к ним появится потенциальная возможность внести ошибку в функционирование

взаимосвязанных между собой методов класса. Поэтому, общим принципом является следующее: «Чем меньше открытых данных о классе используется в программе, тем лучше». Уменьшение количества публичных членов и методов позволит минимизировать количество ошибок. Желательно, чтобы все члены класса были закрытыми и тогда невозможно будет обращаться к членам класса непосредственно с помощью оператора «.» Количество открытых методов также следует минимизировать.

Если в описании элементов класса отсутствует указание метода доступа, то члены и методы считаются закрытыми (`private`). Принято описывать методы за пределами класса.

ЗАДАЧА 10.2. Изменим рассмотренный ранее пример класса `complex`. Добавим метод `vvod`, предназначенный для ввода действительной и мнимой части числа, члены класса и метод `show_complex` сделаем закрытыми, а остальные методы открытыми. Текст программы будет иметь вид:

```
#include <iostream>
#include <string>
#include <math.h>
#define PI 3.14159
using namespace std;
class complex {
//Открытые методы.
public:
void vvod();
double modul();
double argument();
//Закрытые члены и методы.
private:
double x;
double y;
void show_complex();
};
//Описание открытого метода vvod класса complex.
void complex::vvod()
{
cout<<"Vvedite x\t";
cin>>x;
cout<<"Vvedite y\t";
cin>>y;
// Вызов закрытого метода show_complex
//из открытого метода vvod.
show_complex();
}
//Описание открытого метода modul класса complex.
```

```

double complex::modul()
{
return pow(x*x+y*y,0.5);
}
//Описание открытого метода argument класса complex.
double complex::argument()
{
return atan2(y,x)*180/PI;
}
//Описание закрытого метода modul класса complex.
void complex::show_complex()
{
if (y>=0)
cout<<x<<"+"<<y<<"i"<<endl;
else cout<<x<<y<<"i"<<endl;
}
int main()
{
complex chislo;
chislo.znach();
cout<<"Modul kompleksnogo chisla="<<chislo.modul();
cout<<endl<<"Argument kompleksnogo
chisla="<<chislo.argument()<<endl;
return 1;
}

```

Результат работы программы:

Vvedite x 3

Vvedite y -1

3-1i

Modul kompleksnogo chisla=3.16228

Argument kompleksnogo chisla=-18.435

В рассмотренном примере показано совместное использование открытых и закрытых элементов класса. Разделение на открытые и закрытые в этом примере несколько искусственное, оно проведено только для иллюстрации механизма совместного использования закрытых или открытых элементов класса. Если попробовать обратиться к методу `show_complex()` или к членам класса `x`, `y` из функции `main`, то компилятор выдаст сообщение об ошибке (доступ к элементам класса запрещен). При создании в программе экземпляра класса, формируется указатель `this`, в котором хранится адрес переменной-экземпляра класса. Указатель `this` выступает в роли указателя на текущий объект.

10.3. Использование конструкторов

В предыдущем примере метод `chislo.vvod()` использовался для присвоения начального значения некоторым членам класса, однако для упрощения процесса инициализации объекта предусмотрена специальная функция, которая называется *конструктором*. Имя конструктора совпадает с именем класса, конструктор запускается автоматически при создании экземпляра класса (при объявлении переменной типа `class`). Функции-конструкторы не возвращают значение, но при описании функции конструктора не следует указывать в качестве возвращаемого значения тип `void`.

Конструктор автоматически запускается на выполнение для каждого экземпляра класса при его описании. Чаще всего конструктор служит для инициализации полей экземпляра класса.

ЗАДАЧА 10.3. Добавим в созданный в предыдущем примере класс `complex` конструктор:

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <math.h>
#define PI 3.14159
using namespace std;
//Объявляем класс complex.
//Внутри класса указаны, только прототипы методов,
//а сами функции описаны за пределами класса.
class complex
{
public:
    //Прототип конструктора класса.
    complex();
    //Прототип метода modul().
    double modul();
    //Прототип метода argument().
    double argument();
private:
    double x;
    double y;
    //Прототип метода show_complex().
    void show_complex();
};
    // Главная функция.
int main()
{
    //Описываем экземпляр класса, при выполнении
    //программы после создания переменной
```

```

        //автоматически вызывает конструктор.
complex chislo;
cout<<"Modul kompleksnogo chisla="<<chislo.modul();
cout<<endl<<"Argument kompleksnogo
chisla="<<chislo.argument()<<endl;
return 1;
}
        //Текст функции конструктор класса complex.
complex::complex()
{
cout<<"Vvedite x\t";
cin>>x;
cout<<"Vvedite y\t";
cin>>y;
show_complex();
}
        //Текст метода modul класса complex.
double complex::modul()
{
return pow(x*x+y*y,0.5);
}
        //Текст метода argument класса complex.
double complex::argument()
{
return atan2(y,x)*180/PI;
}
        //Текст метода show_complex класса complex.
void complex::show_complex()
{
    if (y>=0) cout<<x<<"+"<<y<<"i"<<endl;
    else cout<<x<<y<<"i"<<endl;
}

```

Если члены класса, являются массивами (указателями), то в конструкторе логично предусмотреть не только может взять на выделение памяти для него.

ЗАДАЧА 10.4. С использованием классов решить следующую задачу. Заданы координаты n точек в k -мерном пространстве. Найти точки, расстояние между которыми наибольшее и наименьшее.

Для решения задачи создадим класс `prostr`.

Члены класса:

- `int n` – количество точек;
- `int k` – размерность пространства;
- `double **a` – матрица, в которой будут храниться координаты точек, `a[i][j]` - i -я координата точки с номером j .

- `double min` – минимальное расстояние между точками в `k`-мерном пространстве;
- `double max` – максимальное расстояние между точками в `k`-мерном пространстве;
- `int imin, int jmin` – точки, расстояние между которыми минимально;
- `int imax, int jmax` – точки, расстояние между которыми максимально.

Методы класса:

- `prostr()` – конструктор класса, в котором определяются `n` – количество точек, `k` – размерность пространства, выделяется память для матрицы `a` координат точки и вводятся координаты точек;
- `poisk_max()` – функция нахождения точек, расстояние между которыми наибольшее;
- `poisk_min()` – функция нахождения точек, расстояние между которыми наименьшее;
- `vivod_result()` – функция вывода результатов: значений `min`, `max`, `imin`, `jmin`, `imax`, `jmax`;
- `delete_a()` – освобождение памяти выделенной для матрицы `a`.

В главной программе необходимо будет описать экземпляр класса и последовательно вызвать методы `poisk_min()`, `poisk_max()`, `vivod_result()`, `delete_a()`.

Текст программы:

```
#include <iostream>
#include <math.h>
using namespace std;
    //Описываем класс prostr
class prostr{
    //Открытые методы класса.
public:
    //Конструктор класса
    prostr();
    double poisk_min();
    double poisk_max();
    int vivod_result();
    int delete_a();
    //Все члены класса – закрытые.
private:
    int n;
    int k;
    double **a;
    double min;
```



```

double max;
int imin;
int jmin;
int imax;
int jmax;
};
    //Главная функция
void main()
{
    //Описание переменной - экземпляра класса prostr.
prostr x;
    //Вызов метода poisk_max для поиска
    //максимального расстояния между
    //точками в k-мерном пространстве;
x.poisk_max();
    //Вызов метода poisk_min для поиска
    //максимального расстояния между
    //точками в k-мерном пространстве;
x.poisk_min();
    //Вызов метода vivod_result для вывода результатов
x.vivod_result();
    //Вызов функции delete_a.
x.delete_a();
}
    //Текст функции конструктор класса prostr.
prostr::prostr()
{
int i,j;
cout<<"Vvedite razmernost prostrantva ";
cin>>k;
    cout<<"Vvedite kolichestvo toчек ";
    cin>>n;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"Vvedite koordinati "<<j<<" tochki"<<endl;
for(i=0;i<k;i++)
cin>>a[i][j];
}
}
    //Текст метода poisk_max класса prostr.
double prostr::poisk_max()

```

```

{
int i,j,l;
double s;
for(max=0,l=0;l<k;l++)
max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5);
imax=0;jmax=1;
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
if (s>max)
{
max=s;
imax=i;
jmax=j;
}
}
return 0;
}

//Текст метода poisk_min класса prostr.
double prostr::poisk_min()
{
int i,j,l;
double s;
for(min=0,l=0;l<k;l++)
min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min,0.5);
imin=0;jmin=1;
for(i=0;i<k;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
if (s<min)
{
min=s;
imin=i;
jmin=j;
}
}
}

```

```

return 0;
}
    //Текст метода vivod_result класса prostr.
int prostr::vivod_result()
{
int i,j;
for(i=0;i<k;cout<<endl,i++)
for (j=0;j<n;j++)
cout<<a[i][j]<<"\t";
cout<<"max="<<max<<"\t nomera "<<imax<<"\t"<<jmax<<endl;
cout<<"min="<<min<<"\t nomera "<<imin<<"\t"<<jmin<<endl;
return 0;
}
    //Текст функции деструктор класса prostr.
int prostr::delete_a()
{
delete [] a;
return 0;
}

```

Результаты работы программы:

```

Vvedite razmernost prostrantva 2
Vvedite kolichestvo tochk 4
Vvedite koordinati 0 toчки
1 2
Vvedite koordinati 1 toчки
2 3
Vvedite koordinati 2 toчки
4 5
Vvedite koordinati 3 toчки
-7 -9
1 2 4 -7
2 3 5 -9
max=17.8045          nomera 2 3
min=1.41421         nomera 0 1

```

Также как и любые другие функции, конструкторы могут *перегружаться*. Перепишем предыдущий пример, добавив в него перегружаемый конструктор, в который можно передавать значения n и k. В этом примере экземпляр класса можно описывать например так `prostr x`; в этом случае конструктор вызывается без параметров, или так `prostr x (3,5)`; в этом случае вызывается перегружаемый конструктор с параметрами.

```

#include <iostream>
#include <math.h>
using namespace std;

```

```

class prostr{
public:
prostr(int,int);
prostr();
double poisk_min();
double poisk_max();
int vivod_result();
int delete_a();
private:
int n;
int k;
double **a;
double min;
double max;
int imin;
int jmin;
int imax;
int jmax;
};
void main()
{
    //Можно вызывать конструктор с параметрами
    prostr x(3,5);
    //или без prostr x; в этом случае будет вызываться
    //тот же конструктор, что и в предыдущем примере.
    x.poisk_max();
    x.poisk_min();
    x.vivod_result();
    x.delete_a();
}
prostr::prostr()
{
int i,j;
cout<<"Vvedite razmernost prostrantva";
cin>>k;
cout<<"Vvedite kolichestvo toчек ";
cin>>n;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"Vvedite koordinati "<<j<<" tochki"<<endl;
for(i=0;i<k;i++)

```

```

cin>>a[i][j];
}
}
//Текст второго конструктора
//Нельзя в качестве формальных параметров
//конструктора использовать переменные n и k,
//потому что это имена членов класса.
//Если в качестве формальных параметров указать n и k,
//то внутри конструктора будут использоваться локальные
//переменные n и k, но при этом члены класса prost n и k
//будут не определены.
prostr::prostr(int k1, int n1)
    //Входными параметрами являются размерность
    //пространства n1 и количество
    //точек в пространстве k1.
{
int i,j;
    //Присваиваем членам класса n и k значения
    //входных параметров конструктора
k=k1;
n=n1;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"Vvedite koordinati "<<j<<"tochki"<<endl;
for(i=0;i<k;i++)
cin>>a[i][j];
}}
double prostr::poisk_max()
{
int i,j,l;
double s;
for(max=0,l=0;l<k;l++)
max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5);
imax=0;jmax=1;
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
}
}
}
}

```

```

if (s>max)
{
max=s;
imax=i;
jmax=j;
}
}
return 0;
}
double prostr::poisk_min()
{
int i,j,l;
double s;
for(min=0,l=0;l<k;l++)
min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min,0.5);
imin=0;jmin=1;
for(i=0;i<k;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
if (s<min)
{
min=s;
imin=i;
jmin=j;
}
}
return 0;
}
int prostr::vivod_result()
{
int i,j;
for(i=0;i<k;cout<<endl,i++)
for (j=0;j<n;j++)
cout<<a[i][j]<<"\t";
cout<<"max="<<max<<"\t nomera " <<imax<<"\t"<<jmax<<endl;
cout<<"min="<<min<<"\t nomera " <<imin<<"\t"<<jmin<<endl;
return 0;
}
int prostr::delete_a()
{

```

```

delete [] a;
return 0;
}

```

Теперь рассмотрим, как можно использовать *параметры по умолчанию* в конструкторе. Оставим в нашей задаче только конструктор с параметрами (n_1 и k_1), но сделаем их по умолчанию равными 2 и 10 соответственно. В этом случае, при описании экземпляра класса без параметров n по умолчанию будет равно 2, а k – 10. Однако можно описать экземпляр класса, передав в него любые значения n и k . Например, так: `prostr x(3, 5);`

```

#include <iostream>
#include <math.h>
using namespace std;
class prostr{
public:
    //Прототип конструктора с параметрами
    //по умолчанию k=2, n=10.
    //В качестве имен формальных параметров конструктора
    //не могут быть выбраны переменные n и k, потому
    //что эти имена совпадают с именами членов класса.
    prostr(int k1=2,int n1=10);
    double poisk_min();
    double poisk_max();
    int vivod_result();
    int delete_a();
private:
    int n;
    int k;
    double **a;
    double min;
    double max;
    int imin;
    int jmin;
    int imax;
    int jmax;
};
void main()
{
    //Экземпляр класса можно описывать так
    prostr x(2,3);
    // или так - prostr x; в этом случае k=2, n=10.
    x.poisk_max();
    x.poisk_min();
    x.vivod_result();
}

```

```

x.delete_a();
}
    //Конструктор с параметрами по умолчанию k=2, n=10.
prostr::prostr(int k1, int n1)
{
int i,j;
k=k1;
n=n1;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"Vvedite koordinati "<<j<<" točki"<<endl;
for(i=0;i<k;i++)
cin>>a[i][j];
}
}
double prostr::poisk_max()
{
int i,j,l;
double s;
for(max=0,l=0;l<k;l++)
max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5);
imax=0;jmax=1;
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
if (s>max)
{
max=s;
imax=i;
jmax=j;
}
}
return 0;
}
double prostr::poisk_min()
{
int i,j,l;

```



```

double s;
for(min=0, l=0; l<k; l++)
min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min, 0.5);
imin=0; jmin=1;
for(i=0; i<k; i++)
for(j=i+1; j<n; j++)
{
for(s=0, l=0; l<k; l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s, 0.5);
if (s<min)
{
min=s;
imin=i;
jmin=j;
}
}
return 0;
}
int prostr::vivod_result()
{
int i, j;
for(i=0; i<k; cout<<endl, i++)
for (j=0; j<n; j++)
cout<<a[i][j]<<"\t";
cout<<"max="<<max<<"\tnomera "<<imax<<"\t"<<jmax<<endl;
cout<<"min="<<min<<"\tnomera "<<imin<<"\t"<<jmin<<endl;
return 0;
}
int prostr::delete_a()
{
delete [] a;
return 0;}

```

Еще одним видом конструктора является *конструктор копирования*, который позволяет создать копию экземпляра класса. Это актуально когда необходимы два экземпляра класса с одними и теми же значениями членов класса. Синтаксис заголовка конструктора копирования следующий:

```
public: name_constructor (type_class & name);
```

Здесь

- `name_constructor` – имя конструктора копирования,
- `type_class` – имя класса, передаваемого в конструктор копирования (в конструктор копирования можно передавать только экземпляры класса),

- name – передаваемый в конструктор копирования экземпляр класса.

ЗАДАЧА 10.5. Пусть нужно создать класс SLAU, предназначенный для решения систем линейных алгебраических уравнений.

В классе SLAU будут:

- закрытые члены класса: int n – размерность системы линейных алгебраических уравнений (СЛАУ), double **a – динамическая матрица коэффициентов СЛАУ, double *b – динамический массив коэффициентов правых частей СЛАУ, double *x – динамический массив решения СЛАУ, int pr – переменная, в которой будет храниться: 0, если существует единственное решения системы, -1 – если система имеет бесконечное множество решений и -2 если система не имеет решения.
- открытые методы класса: конструктор SLAU, конструктор копирования SLAU(SLAU & A1), метод Solve, предназначенный для решения СЛАУ, метод Result – предназначенный для вывода результата решения СЛАУ.

В методе Solve будет реализован метод Гаусса решения систем линейных алгебраических уравнений, рассмотренный в седьмой главе.

При решении СЛАУ методом Гаусса изменяется матрица коэффициентов и массив коэффициентов правых частей. Поэтому возникает проблема, как сохранить значения коэффициентов системы линейных алгебраических уравнений. Поступим следующим образом:

1. Создадим экземпляр класса SLAU.
2. С помощью конструктора копирования создадим копию этого экземпляра.
3. Вызовем методы Solve и Result для копии. Использование копии позволит сохранить матрицу a и массив b в исходном экземпляре класса.

Вначале создадим класс для решения СЛАУ, которому дадим имя SLAU, заголовочный файл будет иметь имя **SLAU.h**, cpp файл – **SLAU.cpp**.

В заголовочном файле **SLAU.h** будет храниться описание класса SLAU, в файле **SLAU.cpp** – реализация методов класса. В основной функции приложения необходимо с помощью директивы include подключить используемый класс (в нашем случае – include <SLAU.h>).

После добавления пустого класса SLAU содержимое файла **SLAU.h** будет таким:

```
#pragma once
class SLAU
{
public:
    SLAU ();
public:
    ~SLAU ();
};
```

Добавим в него необходимые члены и методы класса, после чего содержимое файла **SLAU.h** должно стать таким:

```
#pragma once
class SLAU
{
public:
    SLAU();
    SLAU(SLAU & A1);
    int Solve();
    int Result();
public:
    ~SLAU();
private:
    int n;
    double **a;
    double *b;
    double *x;
    int pr;
};
```

Содержимое файла **SLAU.cpp** первоначально имеет вид:

```
#include "SLAU.h"
SLAU::SLAU(void)
{
}
SLAU::SLAU(void)
{
}
```

Для нормального функционирования конструктора в файл **SLAU.cpp** запишем реализацию всех методов класса. Содержимое **SLAU.cpp** должно стать таким:

```
#include "SLAU.h"
#include <iostream>
#include <math.h>
using namespace std;
//Конструктор класса SLAU
SLAU::SLAU()
{
    int i,j;
    cout<<"n=";
    cin>>n;
    a=new double *[n];
    b=new double [n];
    x=new double [n];
    for(i=0;i<n;i++)
        a[i]=new double [n];
```

```

cout<<"Matrica A";
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        cin>>a[i][j];
cout<<"Massiv B";
for(i=0;i<n;i++)
    cin>>b[i];
}
//Конструктор копирования
SLAU::SLAU(SLAU & A1)
{
    int i,j;
    n=A1.n;
    a=new double *[n];
    b=new double [n];
    x=new double [n];
    for(i=0;i<n;i++)
        a[i]=new double [n];
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            a[i][j]=A1.a[i][j];
    for(i=0;i<n;i++)
        b[i]=A1.b[i];
}
//Функция решения СЛАУ методом Гаусса
int SLAU::Solve()
{
    int i,j,k,r;
    double c,M,max,s;
    for(k=0;k<n;k++)
    {
        max=fabs(a[k][k]);
        r=k;
        for(i=k+1;i<n;i++)
            if (fabs(a[i][k])>max)
            {
                max=fabs(a[i][k]);
                r=i;
            }
        for(j=0;j<n;j++)
        {
            c=a[k][j];
            a[k][j]=a[r][j];
            a[r][j]=c;

```

```

    }
    c=b[k];
    b[k]=b[r];
    b[r]=c;
    for(i=k+1;i<n;i++)
    {
        for(M=a[i][k]/a[k][k],j=k;j<n;j++)
            a[i][j]-=M*a[k][j];
        b[i]-=M*b[k];
    }
}
if (a[n-1][n-1]==0)
    if(b[n-1]==0)
    {
        return -1;
        pr=-1;
    }
    else {
        return -2;
        pr=-2;
    }

else
{
    for(i=n-1;i>=0;i--)
    {
        for(s=0,j=i+1;j<n;j++)
            s+=a[i][j]*x[j];
        x[i]=(b[i]-s)/a[i][i];
    }
    pr=0;
    return 0;
}
}
//Функция вывода корней системы
int SLAU::Result()
{
    int i;
    if (pr==0)
        for(i=0;i<n;i++)
            cout<<x[i]<<"\t";
    else
        if (pr==-1)
            cout<<"Great number of Solution";
}

```

```

        else
            cout<<"No solution";
        cout<<endl;
        return 1;
    }
    SLAU::~SLAU()
    {
    }

```

Главная функция проекта:

```

#include "SLAU.h"
#include <iostream>
using namespace std;
int main()
{
    //Создаем экземпляр класса SLAU h
    //путем вызова конструктора SLAU h;
    //Создаем экземпляр класса SLAU h1 с помощью
    //конструктора копирования.
    //В h1 будет храниться та же система линейных
    // алгебраических уравнений, что и в h.
    SLAU h1(h);
    //Для решения системы h1 вызывается метод Solve.
    h1.Solve();
    //Вывод решения системы h1 с помощью метода Result.
    h1.Result();
}

```

В этой задаче встретился пустой метод с именем `~SLAU()`. Этот метод называется *деструктором*. C++ позволяет построить функцию-деструктор, которая выполняется при уничтожении экземпляра класса, что происходит в следующих случаях:

- при завершении программы или выходе из функции;
- при освобождении памяти, выделенной для экземпляра класса.

Деструктор имеет имя `~имя_класса`, не имеет параметров и не может перегружаться. Если деструктор не определен явно, то будет использоваться стандартный деструктор. Допишем в класс SLAU деструктор `~SLAU()`, который будет освобождать память, выделенную для `a`, `b` и `x` и выводить сообщение об уничтожении объекта.

```

SLAU::~SLAU()
{
    int i;
    delete []b;
    delete []x;
    for(i=0;i<n;i++)

```

```

        delete []a[i];
    delete []a;
        cout<<"Memory is exempt";
}

```

Также как и другие типы, классы могут *объединяться в массивы*.

ЗАДАЧА 10.6. Создать массив экземпляров, рассмотренного в задаче 10.4, класса prostr. В классе prostr оставим конструктор без параметров (значения n и k будут вводиться с экрана дисплея).

```

#include <iostream>
#include <math.h>
using namespace std;
    //Описываем класс prostr
class prostr{
public:
    prostr();
    ~prostr();
    double poisk_min();
    double poisk_max();
    int vivod_result();
    int delete_a();
private:
    int n;
    int k;
    double **a;
    double min;
    double max;
    int imin;
    int jmin;
    int imax;
    int jmax;
};
void main()
{
    int m;
        //Описан массив из трех экземпляров класса,
        //при этом для каждого экземпляра класса
        //(x[0], x[1], x[2])автоматически будет
        //вызываться конструктор. Деструктор тоже будет
        //вызываться автоматически для всех
        //экземпляров класса.
    prostr x[3];
        //Для каждого экземпляра класса вызываем методы

```

```

poisk_max(), poisk_min(),
//vivod_result.
for (m=0;m<3;m++)
{
x[m].poisk_max();
x[m].poisk_min();
x[m].vivod_result();
}
}
//Конструктор класса.
prostr::prostr()
{
int i,j;
cout<<"Vvedite razmernost prostrantva";
cin>>k;
cout<<"Vvedite kolichestvo toчек ";
cin>>n;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"Vvedite koordinati "<<j<<" tochki"<<endl;
for(i=0;i<k;i++)
cin>>a[i][j];
}
}
//Деструктор класса
prostr::~~prostr()
{
int i;
for(i=0;i<k;i++)
delete []a[i];
delete []a;
cout<<"delete object!!!"<<endl;
}
//Метод poisk_max()
double prostr::poisk_max()
{
int i,j,l;
double s;
for(max=0,l=0;l<k;l++)
max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5);
}

```



```

imax=0;jmax=1;
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
if (s>max)
{
max=s;
imax=i;
jmax=j;
}
}
return 0;
}

//Метод поиск_min()
double prostr::поиск_min()
{
int i,j,l;
double s;
for(min=0,l=0;l<k;l++)
min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min,0.5);
imin=0;jmin=1;
for(i=0;i<k;i++)
for(j=i+1;j<n;j++)
{
for(s=0,l=0;l<k;l++)
s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
s=pow(s,0.5);
if (s<min)
{
min=s;
imin=i;
jmin=j;
}
}
return 0;
}

//Метод вывод_result()
int prostr::вывод_result()
{
int i,j;

```

```

for(i=0;i<k;cout<<endl,i++)
for (j=0;j<n;j++)
cout<<a[i][j]<<"\t";
cout<<"max="<<max<<"\tnomera " <<imax<<"\t"<<jmax<<endl;
cout<<"min="<<min<<"\tnomera " <<imin<<"\t"<<jmin<<endl;
return 0;
}

```

Результаты работы программы.

Vvedite razmernost prostrantva3

Vvedite kolichestvo tochek 5

Vvedite koordinati 0 tochki

0 0 0

Vvedite koordinati 1 tochki

1 1 1

Vvedite koordinati 2 tochki

2 2 2

Vvedite koordinati 3 tochki

3 3 3

Vvedite koordinati 4 tochki

0.2 0.3 0.7

Vvedite razmernost prostrantva2

Vvedite kolichestvo tochek 3

Vvedite koordinati 0 tochki

0 0

Vvedite koordinati 1 tochki

2 3

Vvedite koordinati 2 tochki

6 -7

Vvedite razmernost prostrantva2

Vvedite kolichestvo tochek 4

Vvedite koordinati 0 tochki

0 0

Vvedite koordinati 1 tochki

-1 -1

Vvedite koordinati 2 tochki

2 2

Vvedite koordinati 3 tochki

3 3

0 1 2 3 0.2

0 1 2 3 0.3

0 1 2 3 0.7

max=5.19615 nomera 0 3

min=0.787401 nomera 0 4

0 2 6

```

0          3          -7
max=10.7703      nomera 1          2
min=3.60555      nomera 0          1
0          -1       2          3
0          -1       2          3
max=5.65685      nomera 1          3
min=1.41421      nomera 0          1
delete object!!!
delete object!!!
delete object!!!

```

Как и любой другой массив, массив экземпляров класса может быть динамическим. Для работы с динамическими массивами экземпляров класса необходимо:

1. Описать указатель на класс.
2. Определить количество экземпляров класса.
3. С помощью оператора `new` создать динамический массив экземпляров класса.

Перепишем функцию `main` для работы с динамическим массивом экземпляров `prostr`.

```

void main()
{
    int m,g;
    //Описываем указатель на класс prostr.
    prostr *x;
    //Определяем количество элементов в
    //динамическом массиве.
    cout<<"g=";
    cin>>g;
    //Создаем динамический массив из g
    //экземпляров класса prostr.
    x=new prostr[g];
    for(m=0;m<g;m++)
    {
        x[m].poisk_max();
        x[m].poisk_min();
        x[m].vivod_result();
    }
}

```

В C++ существует возможность *перегрузки операции внутри класса*, например, можно добиться того, что операция * при работе с матрицами осуществляла умножение матриц, а при работе с комплексными числами – умножение комплексных чисел.

Для перегрузки операций внутри класса нужно написать специальную функцию – метод класса. При перегрузке операций следует помнить следующее:

- при перегрузке нельзя изменить приоритет операций;
- нельзя изменить тип операции (из унарной операции нельзя сделать бинарную или наоборот);
- перегруженная операция является членом класса и может использоваться только в выражениях с объектами своего класса;
- нельзя создавать новые операции;
- запрещено перегружать операции: `.` (доступ к членам класса), унарную операцию `*` (значение по адресу указателя), `::` (расширение области видимости), `?:` (операция if);
- допустима перегрузка следующих операций: `+`, `-`, `*`, `/`, `%`, `=`, `<`, `>`, `+=`, `-=`, `*=`, `/=`, `<<`, `>>`, `&&`, `||`, `++`, `--`, `()`, `[]`, `new`, `delete`.

Для перегрузки бинарной операции внутри класса необходимо создать функцию-метод:

```
type operator symbols(type1 parametr)
{
    операторы;
}
```

здесь

- `type` – тип возвращаемого операцией значения,
- `operator` – служебное слово,
- `symbols` – перегружая операция,
- `type1` – тип второго операнда, первым операндом является экземпляр класса,
- `parametr` – имя переменной второго операнда.

ЗАДАЧА 10.7. Создать класс для работы с комплексными числами, в котором перегрузить операции сложения и вычитания.

Текст программы:

```
#include <iostream>
using namespace std;
//Класс комплексное число complex.
class complex {
public:
    //Конструктор класса
    complex(bool pr=true);
    //Метод, реализующий перегрузку операции сложения.
```

```

complex operator+(complex M);
    //Метод, реализующий перегрузку операции вычитания.
complex operator-(complex M);
    //Действительная часть комплексного числа.
    float x;
    //Мнимая часть комплексного числа.
    float y;
    //Метод вывода комплексного числа на экран.
    void show_complex();
};
int main()
{
complex chislo1, chislo2, chislo4(false), chislo3(false);
    //Для сложения двух комплексных чисел достаточно
    //использовать операцию +.
    //В классе complex + перегружен для выполнения
    //сложения комплексных чисел.
chislo3=chislo1+chislo2;
cout<<"chislo3=";
chislo3.show_complex();
    //Для вычитания двух комплексных чисел достаточно
    //использовать операцию -.
    //В классе complex - перегружен для выполнения
    //вычитания комплексных чисел.
chislo4=chislo1-chislo2;
cout<<"chislo4=";
chislo4.show_complex();
return 1;
}
//Конструктор класса complex, с логическим параметром
//(true - по умолчанию), если параметр равен true,
//то в конструкторе будет запрашиваться
//действительная и мнимая часть числа,
//если же параметр конструктора равен 0,
//то будет создаваться комплексное число с нулевой
//действительной и мнимой частью.
//Комплексные числа с нулевой действительной
//и мнимой частью можно использоваться для создания
//переменных, в которых будет храниться результат
//действий с комплексными числами.
complex::complex(bool pr)
{
    if (pr)
    {

```

```

        cout<<"Vvedite x\t";
        cin>>x;
        cout<<"Vvedite y\t";
        cin>>y;
        show_complex();
    }
    else{x=0;y=0;}
}
//Метод вывода комплексного числа на экран.
void complex::show_complex()
{
if (y>=0) cout<<x<<"+"<<y<<"i"<<endl;
    else cout<<x<<y<<"i"<<endl;
}
//Метод реализующий перегрузку операции сложения.
//Результатом этой функции будет новое
//комплексное число.
complex complex::operator+(complex M)
{
    //Создаем комплексное число temp,
    //в котором будет храниться результат
    //сложения двух комплексных чисел.
    complex temp(false);
    //Действительная часть нового комплексного
    //числа формируется, как результат сложения
    //действительной части первого и второго операнда,
    //первым операндом является текущий класс,
    //вторым - передаваемое в функцию
    //комплексное число M.
    temp.x=x+M.x;
//Мнимая часть нового комплексного числа формируется,
//как результат сложения мнимой части первого и
//второго операнда.
    temp.y=y+M.y;
    //Возвращаем сумму двух комплексных чисел,
    //в качестве результата.
    return temp;
}
//Метод реализующий перегрузку операции вычитания.
complex operator-(complex M)
{
    complex temp(false);
    //Действительная часть нового комплексного числа
    //формируется, как результат вычитания действительной

```

```

    //части первого и второго операнда,
    //первым операндом является текущий класс,
    //вторым - передаваемое в
    //функцию комплексное число М.
    temp.x=x-M.x;
//Мнимая часть нового комплексного числа формируется,
//как результат вычитания мнимой части
//первого и второго операнда.
    temp.y=y-M.y;
//Возвращаем разность двух комплексных чисел,
//в качестве результата.
    return temp;};

```

В С++ есть унарные операции ++ и --. Действие этих операций отличается при расположении слева и справа от операнда. Рассмотрим, как реализуется перегрузка операции ++x и x++ на примере класса комплексных чисел.

ЗАДАЧА 10.8. Пусть операция ++x увеличивает действительную и мнимую часть комплексного числа x на 1, а x++ увеличивает на 1 только действительную часть комплексного числа x.

```

#include <iostream>
using namespace std;
class complex {
public:
    //Конструктор класса.
    complex(bool pr=true)
{
    if (pr)
    {
        cout<<"Vvedite x\t";
        cin>>x;
        cout<<"Vvedite y\t";
        cin>>y;
        show_complex();
    };
    //Функция, перегружающая оператор ++x,
    //в этом случае это метод без параметров.
    complex operator++()
    {
        //Увеличиваем действительную и мнимую часть на 1.
        x++;
        y++;
        //Возвращаем текущий класс в качестве
        //результата функции.
        return *this;
    }
};

```

```

    }
    //Функция, перегружающая оператор x++,
    //в этом случае это метод с абстрактным параметром
    //целого типа. Наличие целого типа в скобках
    //говорит только о том, что что перегружается
    //оператор x++, а не ++x.
    complex operator++(int)
    {
        //Увеличиваем действительную и мнимую часть на 1.
        x++;
        return *this;
    }
    //Метод вывода комплексного числа на экран.
    void show_complex()
{
    if (y>=0)
        cout<<x<<"+"<<y<<"i"<<endl;
    else
        cout<<x<<y<<"i"<<endl;
};

    float x;
    float y;
};
int main()
{
    //Комплексное число chislo2;
    complex chislo2;
    //Увеличиваем chislo2 на 1, вызвав метод complex
operator++().
    ++chislo2;
    //Вывод комплексного числа.
    cout<<"++chislo2=";
    chislo2.show_complex();
    //Увеличиваем chislo2 на 1, вызвав метод complex
operator++(int).
    chislo2++;
    //Вывод комплексного числа.
    cout<<"chislo2++=";
    chislo2.show_complex();
    return 1;
}

```

Результаты работы программы представлены ниже.

```

Vvedite x      6
Vvedite y      8

```



```
6+8i
++chislo2=7+9i
chislo2++=8+9i
```

Как видно из результатов, были по-разному перегружены операторы ++x и x++.
ЗАДАЧА 10.9. Создать класс для работы с матрицами, перегрузив операции сложения, вычитания и умножения.

```
#include <iostream>
using namespace std;
class matrix {
public:
    //Конструктор
    matrix (int k, int p, bool pr=true);
    //Перегрузка операции + для выполнения
    //операции сложения матриц.
    matrix operator+(matrix M);
    //Перегрузка операции + для выполнения операции
    //добавления к матрице вещественного числа.
    matrix operator+(float M);
    //Перегрузка операции - для выполнения операции
    //вычитания матриц.
    matrix operator-(matrix M);
    //Перегрузка операции - для выполнения операции
    //вычитания из матрицы вещественного числа.
    matrix operator-(float M);
    //Перегрузка операции * для выполнения операции
    //умножения матриц.
    matrix operator*(matrix M);
    //Перегрузка операции * для выполнения операции
    //умножения матрицы на вещественное число.
    matrix operator*(float M);
    ~matrix();
    //Метод вывода матрицы на экран построчно.
    void show_matrix();
private:
    //Двойной указатель для хранения матрицы.
    float **a;
    //Число строк в матрице.
    int n;
    //Число столбцов в матрице.
    int m;
    //Логическая переменная fl принимает значение false,
    //если матрицу сформировать не удалось.
    bool fl;
};
```

```

        //Главная функция.
int main()
{
matrix a1(3,3), b1(3,3), c1(3,3,false);
c1=a1+b1;
c1.show_matrix();
c1=a1-100;
c1.show_matrix();
c1=b1*5;
c1.show_matrix();
c1=a1*b1;
c1.show_matrix();

}
        //Конструктор.
matrix::matrix(int k, int p,bool pr)
{
int i,j;
n=k;
m=p;
fl=true;
a=new float*[n];
for(i=0;i<n;i++)
a[i]=new float[m];
if (pr)
{
cout<<"Matrix"<<endl;
for(i=0;i<n;i++)
for(j=0;j<m;j++)
cin>>a[i][j];
}
else
for(i=0;i<n;i++)
for(j=0;j<m;j++)
a[i][j]=0;
}
        //Перегрузка операции сложения матриц.
matrix matrix::operator+(matrix M)
{
int i,j;
        //Временная матрица temp для хранения результата //
сложения двух матриц.
matrix temp(n,m,false);
        //Если обе матрицы одинакового размера, то

```

```

if ((n==M.n) && (m==M.m))
{
    //формируем матрицу temp, как сумму матриц.
for(i=0;i<n;i++)
for(j=0;j<m;j++)
temp.a[i][j]=a[i][j]+M.a[i][j];
}
else
    //Если размеры матриц не совпадают, то fl=false //
(результатирующую матрицу сформировать не удалось).
temp.fl=false;
    //Возвращаем матрицу temp, как результат операции.
return temp;
}
    //Перегрузка операции + для выполнения операции //
добавления к матрице вещественного числа.
matrix matrix::operator+(float M)
{
int i,j;
matrix temp(n,m,false);
for(i=0;i<n;i++)
for(j=0;j<m;j++)
temp.a[i][j]=a[i][j]+M;
    //Возвращаем матрицу temp, как результат операции.
return temp;
}
    //Перегрузка операции вычитания матриц.
matrix matrix::operator-(matrix M)
{
int i,j;
    //Временная матрица temp для хранения результата //
вычитания двух матриц.
matrix temp(n,m,false);
    //Если обе матрицы одинакового размера, то
if ((n==M.n) && (m==M.m))
{
    //формируем матрицу temp, как разность матриц.
for(i=0;i<n;i++)
for(j=0;j<m;j++)
temp.a[i][j]=a[i][j]-M.a[i][j];
}
else
    //Если размеры матриц не совпадают, то fl=false //
(результатирующую матрицу сформировать не удалось).

```

```

temp.fl=false;
    //Возвращаем матрицу temp, как результат операции.
return temp;
}
    //Перегрузка операции - для выполнения операции
//вычитания из матрицы вещественного числа.
matrix matrix::operator-(float M)
{
int i,j;
matrix temp(n,m,false);
for(i=0;i<n;i++)
for(j=0;j<m;j++)
temp.a[i][j]=a[i][j]-M;
    //Возвращаем матрицу temp, как результат операции.
return temp;
}
    //Перегрузка операции умножения матриц
matrix matrix::operator*(matrix M)
{
int i,j,k;
    //Временная матрица temp для хранения результата
//умножения двух матриц.
matrix temp(n,M.m,false);
//Если количество столбцов в первой матрицы совпадает
//с количеством строк во второй матрицы, то
if ((m==M.n))
{
    //выполняем умножение матриц
for(i=0;i<n;i++)
for(j=0;j<M.m;j++)
for(k=0,temp.a[i][j]=0;k<m;k++)
temp.a[i][j]+=a[i][k]*M.a[k][j];
}
    //Если количество столбцов в первой матрицы не
//совпадает с количеством строк во второй матрице,
//то fl=false (результатирующую матрицу
//сформировать не удалось).
else
temp.fl=false;
    //Возвращаем матрицу temp, как результат операции
return temp;
}
    //Перегрузка операции * для выполнения операции
//умножения матрицы на вещественное число.

```

```

matrix matrix::operator*(float M)
{
    int i,j;
    matrix temp(n,m,false);
    for(i=0;i<n;i++)
    for(j=0;j<m;j++)
    temp.a[i][j]=a[i][j]*M;
    //Возвращаем матрицу temp, как результат операции.
    return temp;
}
matrix::~~matrix()
{
}
//Метод вывода матрицы.
void matrix::show_matrix()
{
    int i,j;
    //Если матрица сформирована, то выводим ее на экран.
    if (fl)
    {
        cout<<"Matrix"<<endl;
        for(i=0;i<n;cout<<endl,i++)
        for(j=0;j<m;j++)
        cout<<a[i][j]<<"\t";
    }
    //Если матрицу сформировать не удалось, то выводим
    //сообщение об этом на экран.
else
cout<<"No Matrix"<<endl;
}

```

Одной из основных особенностей ООП является возможность наследования. *Наследование* – это способ повторного использования программного обеспечения, при котором новые *производные* классы (наследники) создаются на базе уже существующих *базовых* классов (родителей). При создании новый класс является наследником членов и методов ранее определенного базового класса. Создаваемый путем наследования класс является *производным* (derived class), который в свою очередь может выступать в качестве *базового* класса (based class) для создаваемых классов. Если имена методов производного и базового классов совпадают, то методы производного класса перегружают методы базового класса.

При использовании наследования члены и методы кроме свойств public и private могут иметь свойство protected. Для одиночного класса

описатели `protected` и `private` равносильны. Разница между `protected` и `private` проявляется при наследовании, закрытые члены и методы, объявленные в базовом классе, как `protected`, в производном могут использоваться, как открытые (`public`). *Защищенные*(`protected`) члены и методы являются чем-то промежуточным между `public` и `private`.

При создании производного класса используется следующий синтаксис.

```
class name_derived_class:
type_inheritance base_class
{
// закрытые члены и методы класса
...
public:
// открытые члены и методы класса
...
protected:
// защищенные члены и методы класса
...
};
```

Здесь

- `name_derived_class` – имя создаваемого производного класса,
- `type_inheritance` – способ наследования, возможны следующие способы наследования `public`, `private` и `protected`.
- `base_class` – имя базового типа.

Следует различать тип доступа к элементам в базовом классе и тип наследования.

Типы наследования и типа доступа

Способ доступа	Спецификатор в базовом классе	Доступ в производном классе
<code>private</code>	<code>private</code> <code>protected</code> <code>public</code>	нет <code>private</code> <code>private</code>
<code>protected</code>	<code>private</code> <code>protected</code> <code>public</code>	нет <code>protected</code> <code>protected</code>
<code>public</code>	<code>private</code> <code>protected</code> <code>public</code>	нет <code>protected</code> <code>public</code>

При порождении производного класса из базового, имеющего конструктор, конструктор базового типа необходимо вызывать из конструктора производного класса. Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Если в конструкторе производного класса явный вызов конструктора базового типа отсутствует, то он вызывается автоматически без параметров. В случае нескольких уровней наследования

конструкторы вызываются, начиная с верхнего уровня. В случае нескольких базовых типов их конструкторы вызываются в порядке объявления.

Если в описании класса есть ссылка на описываемый позже класс, то его надо просто объявить с помощью оператора

```
class new_class;
```

Это описание аналогично описанию прототипов функции.

Зачастую при наследовании появляются методы, которые в различных производных классах работают по различным алгоритмам, но имеют одинаковые выходные параметры и возвращаемое значение. Такие методы называются *виртуальными* и описываются с помощью служебного слова `virtual`.

ЗАДАЧА 10.10. Рассмотрим абстрактный базовый класс `figure` (фигура), на базе которого можно построить производные классы для реальных фигур (эллипс, окружность, квадрат, ромб, прямоугольник, треугольник и т.д.). На листинге приведен базовый класс `figure` и производные классы `_circle` (окружность) и `Rectangle` (прямоугольник).

```
#include "stdafx.h"
#include <iostream>
#include <math.h>
#define PI 3.14159
using namespace std;
    //Базовый класс figure.
class figure
{
public:
    //n - количество сторон фигуры, для окружности n=1.
    int n;
    //p - массив длин сторон фигуры,
    //для окружности в p хранится радиус.
    float *p;
    //Конструктор.
    figure();
    //Метод вычисления периметра фигуры.
    float perimetr();
    //Метод вычисления площади фигуры.
    virtual float square();
    //Метод вывода информации о фигуре: ее название,
    //периметр, площадь и т.д.
    virtual void show_parametri();
};
    //Конструктор класса фигуры.
figure::figure()
{cout<<"This is abstract constructor"<<endl;}
```

```

        //Метод вычисления периметра,
        //он будет перегружаться только в классе _circle.
float figure::perimetr()
{int i;
  float psum;
  for(psum=0,i=0;i<n;psum+=p[i],i++);
  return psum;
}
        //Метод вычисления площади, пока он абстрактный,
        //в каждом классе будет перегружаться
        //реальным методом.
float figure::square()
{cout<<"No square abstract figure"<<endl;
  return 0;
}
        //Метод вывода информации о фигуре будет
        //перегружаться в каждом производном классе.
void figure::show_parametri()
{cout<<"Abstract figure";}
        //Производный класс _circle (окружность),
        //основанный на классе figure.
class _circle:public figure
{
public:
    //Конструктор
    _circle();
    //Перегружаемые методы
perimetr(),square(),show_parametri().
    float perimetr();
    virtual float square();
    virtual void show_parametri();
};
        //Производный класс RectAngle (прямоугольник), //
основанный на классе figure.
class RectAngle:public figure
{public:
    //Конструктор.
    RectAngle();
    //Перегружаемые методы square(),show_parametri().
    virtual float square();
    virtual void show_parametri();};
        //Главная функция.
void main()
{_circle RR;

```



```

    RR.show_parametri();
    RectAngle PP;
    PP.show_parametri();
}

//Конструктор класса _circle.
_circle::_circle()
{cout<<"Parametri okruzhnosti"<<endl;
 //В качестве сторон окружности выступает
 //единственный параметр радиус.
 n=1;
 p=new float[n];
 cout<<"Vvedite radius";
 cin>>p[0];}
 //Метод вычисления периметра окружности.
float _circle::perimetr()
{
    return 2*PI*p[0];
}
 //Метод вычисления площади окружности.
float _circle::square()
{
    return PI*p[0]*p[0];
}
 //Метод вывода параметров окружности.
void _circle::show_parametri()
{
    //Вывод сообщения о том, что это окружность.
    cout<<"This is circle"<<endl;
    //Вывод радиуса окружности.
    cout<<"Radius="<<p[0]<<endl;
    //Вывод периметра окружности.
    cout<<"Perimetr="<<perimetr()<<endl;
    //Вывод площади окружности.
    cout<<"Square="<<square()<<endl;
}
 //Конструктор класса RectAngle.
RectAngle::RectAngle()
{
    cout<<"Parametri rectangle"<<endl;
    //Количество сторон =4.
    n=4;
    p=new float[n];
    //Ввод длин сторон прямоугольника.
    cout<<"Vvedite dlini storon";
}

```

```

    cin>>p[0]>>p[1];
    p[2]=p[0];
    p[3]=p[1];
}
    //Метод вычисления площади прямоугольника.
float Rectangle::square()
{
    return p[0]*p[1];
}
    //Метод вывода параметров прямоугольника.
void Rectangle::show_parametri()
{
    //Вывод сообщения о том, что это прямоугольник.
    cout<<"This is Rectangle"<<endl;
    //Вывод длин сторон прямоугольника.
    cout<<"a="<<p[0]<<" b="<<p[1]<<endl;
    //Вывод периметра прямоугольника.
    //Классе Rectangle вызывает метод perimetr()
    //базового класса (figure).
    cout<<"Perimetr="<<perimetr()<<endl;
    //Вывод площади прямоугольника.
    cout<<"Square="<<square()<<endl;
}

```

Результаты работы программы.

```

This is abstract constructor
Parametri okruzhnosti
Vvedite radius 5
This is circle
Radius=5
Perimetr=31.4159
Square=78.5397
This is abstract constructor
Parametri rectangle
Vvedite dlini storon3 7
This is Rectangle
a=3 b=7
Perimetr=20
Square=21

```