ЛЕКЦИЯ 10.

Тема: Массивы и указатели

Вопросы

1. Ввод-вывод двумерного массива.
2. Инициализация массива случайными числами.
3. Обработка матриц.
4. Указатели.
5. Динамическое выделение памяти.
6. Динамические массивы.
7. Массивы – параметры.

Ввод-вывод элементов двумерного массива

Пример10.1.1. ***Ввод-вывод элементов двумерного массива (в стиле С).***

```
#include <iostream>
#include <stdio.h>
int main()
{setlocale(LC_ALL,".1251");
 const int nm=20; //макс размер матрицы
 int AB [nm][nm],m,n;
 printf("Размерность матрицы?\n");
 scanf("%d%d",&n,&m);
 printf("Элементы матрицы?\n");
 for (int i=0;i<n;i++)
    for (int j=0;j<m;j++)
       scanf("%d",&AB[i][j]);

 //вывод матрицы
 printf ("Введенная матрица\n");
 for (int i=0;i<n;i++)
 {
    for (int j=0;j<m;j++)
        printf("%5d", AB[i][j]);
    printf("\n");
 }
 return 0;
}
```

Пример 10.1.2. ***Ввод-вывод элементов двумерного массива (в стиле C++)***

```
#include <iostream>
int main()
{setlocale(LC_ALL,".1251");
 const int nm=20; //макс размер матрицы
int AB [nm][nm],m,n;
 cout << "Размерность матрицы?" << endl;
 cin >> n >> m;
 cout << "Элементы матрицы?\n";
 for (int i=0;i<n;i++)
    for (int j=0;j<m;j++)
       cin >> AB[i][j];
 //вывод матрицы
 cout << "Введенная матрица" <<endl;
 for (int i=0;i<n;i++)
 {
    for (int j=0;j<m;j++)
        cout << AB[i][j] << " ";
    cout << endl;
 }
 return 0;
}
```

Генерация элементов матрицы с помощью датчика случайных чисел

Пример 10.2.

```
/*генерация элементов матрицы с помощью датчика случайных
чисел*/
…
#include <time.h>
int main()
…
srand(time(0));
…
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
     AB[i][j] = rand()%100;
…
```

Обработка матриц

Пример 10.3.

```
int main()
/*сумма элементов верхнего правого треугольника матрицы*/
{
 const   int nmax = 10; /*максимальный размер матрицы*/
 float   a[nmax][nmax];
 int n,i,j;
 float sum=0.0;
 cout << "введите размерность\n";
 cin >> n;
 cout <<
     "введите матрицу по строкам\n";
 for (i=0; i<n; i++)
    for (j=0; j<n; j++)
     cin >> a[i][j];
 for (i=0; i<n; i++)
    for (j=i; j<n; j++)
       sum = sum + a[i][j];
 cout << "sum= " << sum << endl;
 return 0;
}
```

Пример 10.4. Построение треугольника Паскаля

вид 1	*вид 2*
1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 1 5 10 10 5 1 1 6 15 20 15 6 1 1 7 21 35 35 21 7 1 1 8 28 56 70 56 28 8 1	1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 1 5 10 10 5 1 1 6 15 20 15 6 1 1 7 21 35 35 21 7 1 1 8 28 56 70 56 28 8 1

```
//треугольник Паскаля
#include <stdio.h>
int main()
{
 const int nmax=10;
 int n,i,j;
 int Ma[nmax][nmax];
 printf("Dimension? ");
 scanf("%d",&n);
 Ma[0][0]=1;
 for (i=1; i<n; i++)
 {
   Ma[i][0]=1;
   Ma[i][i]=1;
   for (j=1; j<i; j++)
    Ma[i][j]=Ma[i-1][j-1]+Ma[i-1][j];
 }
 printf("Result\n"); //вид2
 for (i=0; i<n; i++)
 {
  for (j=0; j<=i; j++)
    printf("%5d", Ma[i][j]);
  printf("\n");
 }
 return 0;
}
```

Упр. Выведите результат в виде1.

Указатели

Одна из наиболее мощных возможностей языка C++ - это непосредственный доступ к памяти при помощи указателей.

Указатель – это переменная, которая содержит адрес другой переменной (адрес области в оперативной памяти).

Оперативная память состоит из пронумерованных ячеек. Каждая переменная занимает одну или несколько последовательных ячеек.

Переменная X **`(int X=25;)`** занимает 4 байта (например 102-105). Адресом переменной X является адрес ее первого байта, в данном случае 102, который можно будет присвоить указателю.

Формат описания указателей:

```
тип *имя [ = инициализатор ];
```

```
int X=25, *px;
```
X – переменная типа **int,**
px – указатель на переменную типа **int.**
Указатель всегда связан с некоторым базовым типом. Базовый тип не может быть ссылкой, битовым полем и может быть только объявлен, но не определен.

Для манипуляции с адресами используются две основных операции: ***разадресации*** (***разыменования***) и ***получения адреса***.

Операция разыменования имеет вид
***указатель**
и позволяет обратиться к содержимому памяти, адрес которой хранится в указателе.

Операция получения адреса, имеет вид
&переменная

Она возвращает адрес конкретной переменной.
```
int *a;
```
можно прочитать двояко: «а является указателем на тип **int**» и «безымянная переменная, адрес которой хранится в а, имеет тип **int**».

Второе прочтение является более правильным.

Запись операции разыменования схожа с определением указателей.
```
int *px, X=25;
```
Здесь определена переменная **X** типа **int** и указатель **px** на этот тип. Выполним:
```
px=&X;//в px адрес переменной X
*px=100;//значение X стало 100
int *p1, a, b=10, *p2;
```
Здесь определены две переменных типа **int**, одна из которых инициализирована, и два указателя на этот тип).

Существуют ***абстрактные*** указатели, для которых в качестве базового используется тип **void**.

Абстрактному указателю можно присвоить значение указателя на любой тип, а также сравнивать его с любым указателем. Однако перед выполнением каких-либо действий с областью памяти, адрес которой содержится в абстрактном указателе, необходимо выполнить операцию преобразования типов.

Инициализация указателей, присваивание им новых значений, может быть выполнена несколькими способами:

1. Присваивание указателю адреса уже существующего объекта, например:

```
char c;
char *pc = &c;
```

2. Присваивание указателю значения другого указателя, например:

```
char c;
char *pc1 = &c, *pc2 = pc1;
```

3. Присваивание указателю адреса памяти в явном виде.

```
char *pc = (char *)0x000012D4;
```

В этом примере указана операция преобразования типа в виде, стандартном для C++.

4. Присваивание указателю специального пустого значения NULL:

```
char *pc = NULL;
```

Значение NULL гарантирует, что объектов по этому адресу не будет, и при попытке разыменования пустого указателя возникнет исключительная ситуация. Вместо NULL можно использовать значение 0 (учитывая то, что указатели по сути совместимы с целочисленными типами). Однако для того, чтобы сделать программу более понятной, рекомендуется использовать именно идентификатор NULL.

При неаккуратном использовании динамической памяти возможно появление «мусора», т.е. такой памяти, которая считается выделенной программе, но ее адрес по каким-то причинам утерян, и доступ к ней невозможен. Пример появления мусора:

```
int *pi1=new int, *pi2=new int;
*pi1=0;
pi2=pi1;
// хотели написать *pi2 = *pi1  ☹
```

Другая возможность появления мусора связана с тем, что указатели подчиняются общим ограничениям на область действия и время жизни.

```
int my_func()
{
int *pi = new int;
…
} // delete pi написать забыли… ☹
```

при каждом вызове этой функции память под одну переменную типа **int** будет отправляться в мусор.

Поскольку значения указателей, т.е. адреса, по сути являются числами, с ними можно выполнять некоторые арифметические операции: сложение с константой, вычитание, инкремент и декремент. При выполнении этих операций учитывается длина базового типа:

```
int *p;
p++;
```

значение **p** увеличивается не на единицу, а на **sizeof(int)** (арифметические операции над абстрактными указателями недопустимы).

Разность между двумя указателями – это разность их значений, деленная на размер типа в байтах. Суммирование двух указателей не допускается.

При изменении значений указателя не производится никакого контроля за тем, чтобы новый адрес памяти был легально выделен программе. Так, записав последовательность операторов

```
int *pv = new int;
…
pv++;
*pv=1;
```

изменитсям содержимое области памяти, которая либо не была выделена, либо была выделена для совершенно других целей! Поэтому следует соблюдать особую осторожность при изменении значений указателей.

При записи выражений с указателями следует обращать внимание на приоритет операций и при необходимости расставлять скобки. Так, выражение **(*p)++** увеличивает значение, расположенное по адресу, хранящемуся в p. С другой стороны, выражение ***(p++)** изменяет значение указателя. Тем не менее, оба выражения возвращают старые данные, хранящиеся по старому адресу. Поэтому, записывая оператор ***p++,** задумайтесь – поймет ли компилятор то, что вы хотели сказать!

(***p++** выполняется в C++ как ***(p++)**)

Константные указатели

В C++ они подразделяются на два типа:

- указатели, значение которых не изменяется в ходе выполнения программы. Они описываются, например, так:

```
int a;
int * const p=&a;
```

В этом случае значение **p** не может быть изменено, и оператор **p++** вызовет ошибку;

- указатели, хранящие адрес константы. Они описываются следующим образом:

```
int a;
const int *p=&a;
```

В этом случае ошибку компиляции вызовет оператор **(*p)++.**

С другой стороны, оператор **a++** считается вполне нормальным.

Динамическое выделение памяти

Выделение динамической памяти для переменной типа `int` можно выполнить двумя способами:

a) с помощью функции **`malloc`**, описанной еще в языке С:

```
int *pi=(int*)malloc(sizeof(int));
```

(функция **`malloc`** возвращает значение типа **`void`** *, и необходимо выполнить преобразование типа)

b) с помощью появившейся в С++ операции **`new`**:

```
int *pi = new int;
```

Освобождение памяти соответственно:

```
free(pi);
delete pi;
```

После освобождения памяти значение указателя не изменяется!!!

Проблемы, возникающие при работе с указателями

1. ***Использование неинициализированных указателей***

Неинициализированный указатель содержит какое-то значение, которое трактуется как адрес памяти. Если случайно окажется, что этот адрес доступен для приложения – последствия непредсказуемы!

2. ***Использование “зависших” указателей***

После освобождения динамической памяти ее адрес остается доступным в программе. Обращение по такому адресу приведет либо к разрушению динамической памяти, либо к доступу к повторно выделенной памяти.

3. ***“Утечка” памяти***

Потеря значения адреса выделенной памяти влечет невозможность доступа к ней (в том числе и освобождения) до окончания работы программы.

4. ***Возможные причины утечки памяти***

Изменение значения указателя, который использовался при выделении памяти;

Выход такого указателя из области своего действия

При неаккуратном использовании динамической памяти возможно появление «мусора», (или «утечки» памяти), т.е. такой памяти, которая считается выделенной программе, но ее адрес по каким-то причинам утерян, и доступ к ней невозможен.

Примеры неправильной работы с памятью

```
int *pi1=new int, *pi2=new int;
*pi1=0;
pi2=pi1;
```

```
// хотели написать *pi2 = *pi1  ☹
// получили утечку памяти
…
(*pi2)++;
// тем самым изменилось и *pi1, но
// этого не видим
…
delete pi1; // пока нормально
delete pi2; // использование зависшего указателя
крах!!!
```

Другая возможность появления мусора связана с тем, что указатели подчиняются общим ограничениям на область действия и время жизни.

```
int my_func()
{
int *pi = new int;
…
} // delete pi написать забыли… ☹
```

при каждом вызове этой функции память под одну переменную типа **int** будет отправляться в мусор.

Поскольку значения указателей, т.е. адреса, по сути являются числами, с ними можно выполнять некоторые арифметические операции: сложение с константой, вычитание, инкремент и декремент. При выполнении этих операций учитывается длина базового типа:

```
int *p;
p++;
```

значение **p** увеличивается не на единицу, а на **sizeof(int)** (арифметические операции над абстрактными указателями недопустимы).

Разность между двумя указателями – это разность их значений, деленная на размер типа в байтах. Суммирование двух указателей не допускается.

При изменении значений указателя не производится никакого контроля за тем, чтобы новый адрес памяти был легально выделен программе.

Пример неправильной работы с памятью (часть 2)

```
int *pv = new int;
…
pv++;
*pv=1;
```

изменится содержимое области памяти, которая либо не была выделена, либо была выделена для совершенно других целей! Поэтому следует соблюдать особую осторожность при изменении значений указателей.

При записи выражений с указателями следует обращать внимание на приоритет операций и при необходимости расставлять скобки. Так, выражение **(*p)++** увеличивает значение, расположенное по адресу, хранящемуся в **p.** С другой стороны, выражение ***(p++)** изменяет значение указателя. Тем не менее, оба выражения возвращают старые данные, хранящиеся по старому адресу. Поэтому, записывая оператор ***p++,** задумайтесь – поймет ли компилятор то, что вы хотели сказать!

(*p++ выполняется в **C++** как ***(p++))**

Динамическое выделение памяти под одномерный массив

Динамическое выделение памяти под массив из **N** элементов типа **int** :

1) с помощью функции **malloc** (предполагается, что значение N известно):

```
int *pi = (int *)malloc(N*sizeof(int));
```

2) с помощью операции **new:**

```
int *pi = new int[N];
```

Освобождение памяти, занятой этим массивом, выполняется соответственно операторами

```
free(pi);
delete [] pi;
```

Обратите внимание на необходимость указания скобок в операции **delete**.

Доступ к i-му элементу:

pi[i] или ***(pi+i)**

```
int a[]={1,2,3,4,5,6,7};
int x;
int *pa;
pa = &a[0];  //pa=a;
x = *pa;
cout<< x;           //1;
int i=4;
cout<< *(pa+4);     //5;
cout<< *(pa+i);     //5;
```

Таким образом, если **pa** указывает на **a[0],** то ***(pa+1)** есть содержимое **a[1], a+i -** адрес **a[i], *(pa+i)** - содержимое **a[i].**

Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы **pa+1** указывал на следующий объект, а **pa+i** - на **i**-й после **pa**.

```
pa = &a[0];
pa = a;
```
pa и **a** имеют одно и то же значение.

Пусть **a** – массив и **pa** указатель:
```
int a[]={1,2,3,4,5,6,7};
int x;
int *pa=a;
```
Для обращения к **i** – тому элементу

a[i] и ***(a+i).**

pa[i] и ***(pa+i).**

Элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие.

Указатель - это переменная, поэтому можно написать **pa=a** или **pa++.** Имя массива *не является переменной*, и записи вроде **a=pa** или **a++** не допускаются.

```
int a[]={1,2,3,4,5,6};
int b[]={1,2,3,4,5,6};
int n, m;
n=&a[5]-&a[3];  //n=2
m=&a[3]-&a[5];  //m=-2
int *p;
p=(a+2);        //p=&a[2];
```

Динамическое выделение памяти под двумерный массив

Двумерный статический массив описывается, например,
```
int a[10][10];
```
Двумерные динамические массивы реализуются через указатели на указатели. Описание
```
int a[10][10];
```
соответствует описанию
```
int (*a)[10];
int **a;
```

Для динамического массива типа **int** указатель **a** содержит адрес одномерного массива указателей типа **int *,** хранящих адреса строк массива (т.е. элементов массива с одинаковым значением первой размерности).

Динамическое выделение памяти под двумерный массив размерности MxN (M строк, N столбцов)

1. Использование функции malloc :

```
int **mas;
mas = (int **) malloc(M * sizeof(int *));
for (int i=0; i<M; i++)
 mas[i] = (int *) malloc(N * sizeof(int));
```

Для корректного освобождения памяти необходимо записать следующий фрагмент:

```
for (i=0; i<M; i++)
     free(mas[i]);
free(mas);
```

2. Использование операций new и delete:

```
int **mas;                    // 1
mas = new int *[M];           // 2
for (int i=0; i<M; i++)       // 3
      mas[i] = new int [N]; // 4
```

1 – объявляется переменная типа <указатель на указатель на int>
2 – выделяется память под массив указателей на строки массива (M строк)
3 – организуется цикл для выделения памяти под каждую строку
4 – каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку из N целых чисел

```
For (int i=0; i<M; i++)
   delete [] mas[i];
delete [] mas;
```

В операции **delete** указываются одни скобки независимо от числа размерностей массива.

При статическом описании массива все строки располагаются в памяти друг за другом, однако при динамическом выделении памяти это совсем не обязательно.

Динамические массивы при создании нельзя инициализировать, и они не обнуляются при создании.

Треугольные динамические матрицы

- Выделение памяти под верхний треугольник квадратной матрицы nxn:

```
int **p;
p = new int *[n];
for (int i=0; i<n; i++)
     p[i] = new int [n-i];
```

- Выделение памяти под нижний треугольник такой же матрицы:

```
int **p;
p = new int *[n];
for (int i=0; i<n; i++)
     p[i] = new int [i+1];
```

Пример 10.6. Динамическая единичная матрица (вариант1).

```
include <iostream.h>
void main ()
{int i, j, n;
 cout << "Enter n"; cin >> n;
 int **matr;
 matr= new int *[n];
     //массив указателей
 if (matr == NULL)
     {cout <<"Error"; return;}
 for (i=0;i<n;i++)
      matr[i]= new int [n];
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      if (i != j)
            *(*(matr+i)+j) =0;
       else
            *(*(matr+i)+j) =1;
for (i=0;i<n;i++)
{
 cout << endl;
 for (int j=0;j<n;j++)
  cout<<*(*(matr+i)+j)<<' '; }
 for (i=0;i<n;i++)
     delete *(matr+i);
 delete [] matr;
}
```

Пример 10.7. ***Динамическая единичная матрица (вариант2).***

```
int i,j,n;
cout << "N";
cin >> n;
float **matr;
matr= (float**)malloc(n*4);
if (matr == NULL)
   {cout <<"error"; return;}
for (i=0;i<n;i++)
    matr[i] =(float*)malloc(n*sizeof (float));
 for (i=0;i<n;i++)
   for (j=0;j<n;j++)
     if (i != j)
          *(*(matr+i)+j) =0;
     else
          *(*(matr+i)+j) =1;
for (i=0;i<n;i++)
   { cout << endl;
     for (int j=0;j<n;j++)
      cout << *(*(matr+i)+j) << ' ';
   }
for (i=0;i<n;i++)
    free(*(matr+i));
free(matr);
cout << endl;
}
```

Одномерные массивы – параметры

Формальный параметр массив можно объявить тремя способами:

- массив с фиксированной длиной;
- безразмерный массив;
- указатель, тип которого будет совпадать с типом элементов массива.

При вызове функции, параметром которой является массив, в качестве фактического параметра указывается имя **массива без индексов.**

Пример 10.8. **Нахождение суммы элементов массива.**

```
int sum1(int a[100], int n)
{
 for (int i=0, int s=0; i<n-1;
     s += a[i++]);
 return s;
```

```
}

int sum2(int a[], int n)
{
 int s=0;
 for (int i=-1;i<n-1;s+=a[++i]);
 return s;
}

int sum3(int *a, int n)
{
 int s=0;
 for (int i=-1;i<n-1;s+=a[++i]);
 return s;
}
int main()
{int  a[30], *b,
 c[]={1,22,35,4,5,7,9,6,44,55,100};
 int  r, n;
 srand(time(0));
 n=rand()%29+1; //размер<30
 b=new int[n];
 for (int i=0; i<n; i++)
  {a[i] = rand()%100;
   b[i] = rand()%10;}
 r1= sum1(a,n);
 r2= sum2(c,sizeof(c)/sizeof(int));
 r3= sum3(b,n);
}
```

Функции, возвращающие указатель на массив

Пример 10.9. ***Слияние двух упорядоченных массивов***

```
#include <iostream>
using namespace std;
int* fun(int n, int* a, int m, int* b)
{  int* x=new int[n+m];
   //массив-результат
int ia=0,ib=0,ix=0;
while ((ia<n)&&(ib<m))
//цикл до конца одного из массивов
         if (a[ia]>b[ib])
```

```
                x[ix++]=b[ib++];
            else x[ix++]=a[ia++];
        if (ia>=n)
            while(ib<m)
                x[ix++]=b[ib++];
        else
            while(ia<n)
                x[ix++]=a[ia++];
        return x;
}

void main()
{setlocale(LC_ALL,".1251");
 int p[]={3,7,8,55};
 int s[]={1,4,6,9,23,34};
 int* h;//указатель на массив-результат
 int kp=sizeof(p)/sizeof(p[0]);
 int ks=sizeof(s)/sizeof(s[0]);
 h=fun(kp,p,ks,s);
 cout<<"результат"<<endl;
 for(int i=0;i<kp+ks;i++)
  cout << "  " << h[i];
 cout<<endl;
 delete[]h;
}
```

Двумерные массивы – параметры

Двумерный массив как формальный параметр можно объявить следующими способами:

1. Двумерный массив с фиксированной длиной;

2. Двумерный массив с фиксированной размерностью первого измерения, т.е. второй размерностью (количество столбцов);

3. Указатель, которому при вызове будет соответствовать адрес первого элемента двумерного массива;

4. Массив указателей на массивы строк;

5. Динамический двумерный массив.

Прототипы соответствующих функций будут иметь следующий вид:

```
1)int f1(int ma[20][30]);
2)int f21(int ma[][30], int n);
```

или

```
  int f22(int ma[][30],int n,  int m);
3)int f3(int *ma, int n, int m);
4)int f4(int *ma[],int n,int m);
5)int f5(int **ma, int n,int m);
```

Варианты I-II.

Пример 10.10. Функция транспонирования квадратной матрицы (варианты 1-2).

```
void transpon(int Ma[10][10],int n)
//void transpon(int Ma[][10],int n)
{int r;
 for (int i=0; i<n-1; i++)
   for (int j=i+1; j<n; j++)
   {
     r=Ma[i][j];
     Ma[i][j]=Ma[j][i];
     Ma[j][i]=r;
   }
}

int main()
{int Matr[10][10];
 int k;
 cin >> k;
 for (int i=0; i<k; i++)
   for (int j=0; j<k; j++)
      Matr[i][j]=rand()%10;
 for (int i=0; i<k; i++)
   {for (int j=0; j<k; j++)
      cout << Matr[i][j] << " ";
    cout << endl;
   }

transpon(Matr,k);
 cout << "---------------\n";
 for (int i=0; i<k; i++)
   {for (int j=0; j<k; j++)
      cout << Matr[i][j] << " ";
    cout << endl;
   }
 return 0;
}
```

В головной программе матрица должна быть описана с фиксированной размерностью с 10 столбцами.

А так как матрица квадратная, то и количество строк тоже 10.

Функцию можно вызывать для любой размерности не превышающей 10.

Пример 10.11. ***Функция ввода элементов матрицы.***

```
void input_matr(int Ma[100][100], int *n, int *m)
{ printf("Dimension? ");
  scanf("%d%d", n , m);
  for (int i=0; i<*n; i++)
   for (int j=0; j<*m; j++)
    scanf("%d",&Ma[i][j]);
}
```

В головной программе двумерный массив должен быть описан с размерностью 100x100, но можно обрабатывать размерности n*m, где n<=100, m<=100.

Вызов функции:

```
int M[100][100];
int n, m;
input_matr(M, &n, &m);
```

Пример 10.12. ***Функция вывода элементов матрицы***

```
void print_ma(int Ma[100][100], int n, int m)
//void print_ma(int Ma[][100], int n,int m);
{for ( int i=0;i<n;i++)
 {for (int j=0; j<=m; j++)
    printf("%5d",Ma[i][j]);
  printf("\n");
 }
}
```

Двумерный массив должен быть описан со второй размерностью =100, например

```
int Ma[10][100],
int Ma1[20][100],
int Ma2[100][100],
```

но можно обрабатывать размерности n*m, где n<=100, m<=100.

```
print_ma(Ma,3,4);
print_ma(Ma1,20,12);
```

Пример 10.13 ***Функция генерации элементов матрицы***

```
void rnd_ma(int Ma[100][100], int n, int m)
//void rnd_ma(int Ma[][100], int n, int m)
{
    srand(time(0));
    for (int i=0; i<n; i++)
     for (int j=0;j<m; j++)
        Ma[i][j] = rand()%100;
}

int sum_vt(int ma[100][100],int n)
{
    int s = 0;
    for (int i = 0; i < n;  ++i)
     for (int j = 0; j < n-i; ++j)
        s += ma[i][j];
    return s;
}
```

Вариант III.

Передача двумерного массива через указатель, которому соответствует адрес первого элемента двумерного массива.

Пример 10.14. ***Функция вывода матрицы***.

```
void print_matr(int *ma,
                int dim1, int dim2)
{for (int i = 0; i<dim1; ++i)
 {for (int j = 0; j<dim2; ++j)
   printf("%d\t",ma[i*dim2+j]);
  printf("\n");
 }
}
```

Если при вызове функции использовать как параметр адрес первого элемента, то в этом случае массив должен быть описан точно такой же размерности, как при вызове.

```
int mb[3][3]={1,2,3,1,2,3,1,2,3};
```

Вызов функции:

```
print_matr(&mb[0][0],3,3);
```

Вариант IV.Массив указателей на массивы строк.

Пример 10.15. ***Функция транспонирования квадратной матрицы .***

```
void transpon(int * p[],int n)
{int r;
 for (int i=0; i<n-1; i++)
   for (int j=i+1; j<n; j++)
   {
     r=p[i][j];
     p[i][j]=p[j][i];
     p[j][i]=r;
   }
}

int main()
{int Matr[4][4]={11,12,13,14,
                 21,22,23,24,
                 32,32,33,34,
                 41,42,43,44};
 //вспомогательный массив указателей
 int *ptr[]={(int *)&Matr[0],
             (int *)&Matr[1],
             (int *)&Matr[2],
             (int *)&Matr[3]};
 int k=4;
 transpon(ptr,k);
 cout << "result\n";
 for (int i=0; i<k; i++)
   {for (int j=0; j<k; j++)
      cout << Matr[i][j] << " ";
    cout << endl;
   }
 return 0;
}
```

В головной программе матрица имеет фиксированные размеры. Чтобы передать ее в функцию как параметр вместо формального параметра со спецификацией **int* ptr []** используется вспомогательный массив указателей, значениями которых будут адреса строк матрицы.

Вариант V. Использование динамического двумерного массива.

Пример 10.16. Транспонирование динамической матрицы.

```
void transpon(int **p,int n)
{int r;
 for (int i=0; i<n-1; i++)
   for (int j=i+1; j<n; j++)
   {
      r=p[i][j];
      p[i][j]=p[j][i];
      p[j][i]=r;
   }
}
int main ()
{int i;
 int k=4;
 int **matr;
 srand(time(0));
 matr= new int *[k]; //массив указателей
 for (i=0;i<k;i++)
     matr[i]= new int [k];
 for (int i=0; i<k; i++)
   for (int j=0; j<k; j++)
     matr[i][j]=rand()%10;
 for (i=0;i<k;i++)
{for (int j=0;j<k;j++)
   cout<<*(*(matr+i)+j)<<" ";
  cout << endl;
 }
 transpon(matr,k);
 cout << "_____"<<endl;
 for (i=0;i<k;i++)
 {for (int j=0;j<k;j++)
   cout<<*(*(matr+i)+j)<<" ";
  cout << endl; }
 for (i=0;i<k;i++)
  delete *(matr+i);
 delete [] matr;
}
```

Пример 10.17. ***Функция вывода динамической матрицы.***

```
void print_maD(int **ma,int n, int m)
{
for (int i=0;i<n;i++)
 {for (int j=0;j<m;j++)   printf("%5d",ma[i][j]);
    //printf("%t",ma[i][j]);
  printf("\n");
 }
}
```

Пример 10.18. ***Перепишем функцию подсчета суммы элементов верхнего левого треугольника матрицы.***

```
int sum_vt(int **ma, int n)
{int s = 0;
 for (int i=0; i<n;  ++i)
    for (int j=0; j< n-i;++j)
        s += ma[i][j];
 return s;
}
//Вызов
int res=sum_vt(ma,n);
```

Передача двумерного массива, сформированного в функции в вызывающую программу

Это можно сделать через указатель на одномерный массив указателей на одномерные массивы известной размерности и заданного типа.

Пример 10.19. ***Динамическая единичная матрица, формируемая в функции.***

```
 int** single_matr(int n);
 {
 int **matr;//вспом указ на матр
 matr=new int *[n];//массив указателей
 if (matr == NULL)
 {cout <<"Error"; return;}
 for (i=0;i<n;i++)
    {matr[i]= new int [n];
     if (matr[i] == NULL)
        {cout <<"Error"; return;}
    }
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      if (i != j)
```

```
            *(*(matr+i)+j)=0;
       else
            *(*(matr+i)+j)=1;
 return matr;
}

int main ()
{int n;
 cout << ”Enter n”;
 cin >> n;
 int **ma;
 ma=single_matr(n);
 for (i=0;i<n;i++)
 {
  cout << endl;
  for (int j=0;j<n;j++)
       cout<<*(*(ma+i)+j)<<” ”;
 }
 for (i=0;i<n;i++)
  delete [] *(ma+i);
 delete [] ma;
}
```

Пример 10.20. ***Создание и инициализация динамической матрицы.***

```
int** MatrInt(int **ma, int *n, int *m)
{ printf( "enter n, m");
 scanf("%d%d",n,m);
  ma= new int *[*n];
  for (int i=0;i<*n;i++)
      ma[i]= new int [*m];
  for ( i=0;i<*n;i++)
     for (int j=0;j<*m;j++)
        ma[i][j]= rand()%2;
  return ma;
}
```

Пример дополнительный.
Построить магический квадрат нечетного порядка.

Магический квадрат - это квадратная матрица, у которой сумма чисел каждого столбца равна сумме элементов каждой строки, а также каждой из двух диагоналей .

Рассмотрим построение магического квадрата нечетного порядка, используя индийский метод Москополуса.

```
#include <stdio.h>
void InitNull(int ma[19][19], int n)
 {
  int i,j;
      for(i=0;i<n;i++)
       for(j=0;j<n;j++)
           ma[i][j]=0;
 }
void main ()
{
 int mk[19][19];
 int  n,k,j,i;
 do
   {printf("enter n");
    scanf("%d",&n);//нечетное
   }
 while  (n % 2==0);
 //Заполняем матрицу нулями
 InitNull (mk,n);
 //Определяем координаты первой заполняемой клетки
 i=(n-2)/2;
 j=n/2;
 k=1;
 while (k<=n*n)
 {
  if ((i==-1)&&(j==-1))//вышли в левый верх угол
    {i=n-1; j=n-1;}
  if (j>=n) j=0;
  if (j<0)  j=n-1;
  if (i>=n) i=0;
  if (i<0)  i=n-1;
  //Если клетка существует и не занята
  if (mk[i][j]==0)
```

```
      {
          mk[i][j]=k;
          i--; j--; k++;
      }
  else //Если клетка существует,но занята
    {j++; i--;}
}
for(i=0;i<n;i++)
 {
    for(j=0;j<n;j++)
     printf("%4d",mk[i][j]);
   printf("\n");
 }
}
```

КОНЕЦ ЛЕКЦИИ