
Лабораторная работа.4. Указатели и массивы

Указатель – это ссылка на объект данных или функцию. Указатели имеют множество применений: объявление «вызова по ссылке» функций, реализация динамических структур данных, такие как связанные списки и деревья, и т.п.

Часто единственным путем управления большими значениями данных, это манипулирование не самими данными, а указателями на эти данные. Для примера, если вам нужно сортировать большое количество огромных объектов, более эффективным будет сортировать список указателей на объекты, чем непосредственное перемещение самих объектов в памяти. Если необходимо передать большой объект в функцию, экономичнее передавать указатель на объект, чем передать содержимое объекта, даже если функция не изменяет содержимое.

Объявление указателя

Декларация(объявление) указателя в общем виде:

```
<тип> *<имя_указателя>;
```

Чтобы использовать адрес памяти, компилятор должен знать, какие данные находятся по этому адресу; в противном случае он не может корректно их интерпретировать (например, одни и те же байты имеют разный смысл в данных типа `double` и `int`).

Поэтому, чтобы, например, объявить указатель, который хранит адрес целого числа, необходимо указать тип – `int`, * – значит, что переменная является указателем, имя – переменной-указателя(`pValue`):

```
int *pValue;
```

Обычно префикс `p(p_)` или `ptr(ptr_)` ставит перед именем переменной (`pValue`, `ptrValue`), он удобен, чтобы всегда понимать, что это указатель.

Инициализация указателя

Чтобы определить адрес (местоположение) переменной в памяти, необходимо поставить знак **&**(амперсанд) перед именем переменной. **&** - оператор адреса, поскольку возвращает адрес переменной в памяти.

```
double x = 10.5;
char *p_x = &x; // Ошибка: не соответствие типа; т.к. нет явного
преобразования.
char *p_x = (char *)&x; // Ок: p_x указывает на первый байт x;
```

Выполним данную программу:

```
double x = 10.5;
char *p_x = (char *)&x;
printf("Address of the first byte x = %p \n", p_x); // %p - для вывода в
формате адреса;
```

В результате мы увидим (см. Рис. 4-1) в консоли значение первого байта переменной x в шестнадцатеричном представлении (у вас адрес может быть другим!).

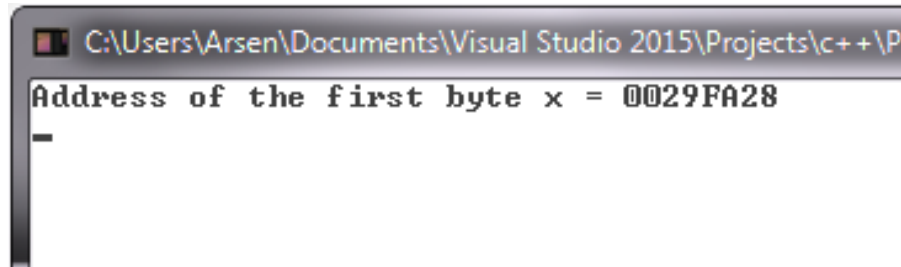


Рис. 4-1 Результат отображения адреса переменной

Операция обратная **&**(взятие адреса) – это операция разыменование указателя - *****.

```
int y = 5;           // y равен 5
int *p_y = &y;       // взятие адреса y
*p_y = 10;           // разыменование указателя и присвоение нового значения
printf("Value of y = %d \n", y);
```

Как видите (Рис. 4-2) через разыменование указателя, который указывал на область памяти переменной y, мы изменили значение y на новое.

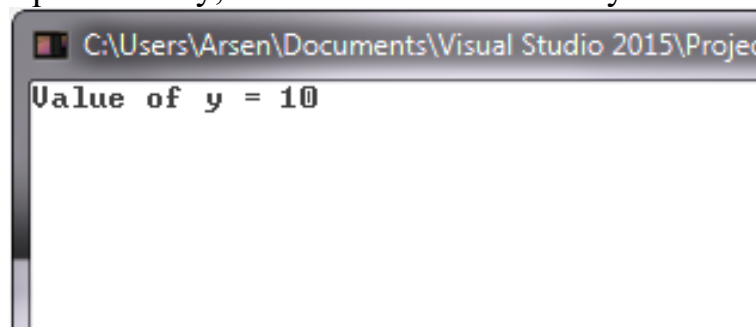


Рис. 4-2 Результат вывода значения переменной через указатель

Рассмотрим пример с определением размера указателя (функция - `sizeof()`):

```
#include <conio.h>
#include <stdio.h>
void main(int argc, char **argv) {
    int x = 100;
    int *p_x = &x;
    double y = 2.3;
    double *p_y = &y;

    printf("%d\n", sizeof(x));
    printf("%d\n", sizeof(p_x));
    printf("%d\n", sizeof(y));
    printf("%d\n", sizeof(p_y));

    _getch(); // у вас возможно просто getch();
}
```

Несмотря на то, что переменные имеют разный тип и размер, указатели на них имеют один размер. Действительно, если указатели хранят адреса, то они должны быть целочисленного типа (Рис. 4-3).

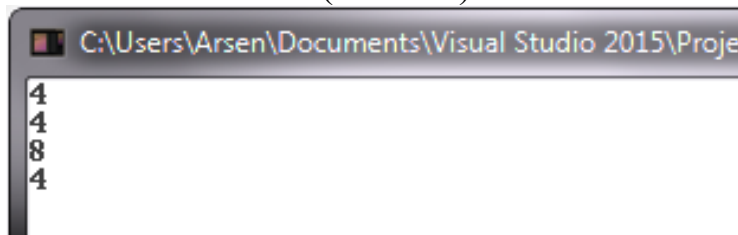


Рис. 4-3 Результат вывода размеров переменных указателей

Зачем указателю нужен тип?

Во-первых, указателю нужен тип для того, чтобы корректно работала операция разыменования (получения содержимого по адресу). Если указатель хранит адрес переменной, необходимо знать, сколько байт нужно взять, начиная от этого адреса, чтобы получить всю переменную.

Во-вторых, указатели поддерживают арифметические операции. Для их выполнения необходимо знать размер.

Т.е. операция $+ N$ сдвигает указатель вперед на $N * \text{sizeof}(\text{тип})$ байт.

Например, если указатель `int *p`; хранит адрес `CC02`, то после `p += 10`; он будет хранить адрес `CC02 + sizeof(int)*10 = CC02 + 28 = CC2A` (в шестнадцатеричном формате).

Указатель на null

Null-указатель – это целочисленная константа со значением 0 или целочисленная константа приведенная к `void` указателю. Макрос `NULL` (определен в `stdlib.h`, `stdio.h` и других) можно использовать для инициализации нулевого указателя. Null указатель может помочь в определении валидности указателя, например, на объект или функцию.

Посмотрим на пример:

```
FILE *fp = fopen("demo.txt", "r");
if (fp == NULL)
{
    // Ошибка: файл demo.txt не был открыт
}
```

Стандартная функция `fopen()` возвращает `NULL` указатель, если возникла ошибка при открытии файла. Таким образом мы можем проверить на правильность (валидность) указателя.

Указатель void

В Си существует особый тип указателей – указатели типа `void` или пустые указатели. Эти указатели используются в том случае, когда тип переменной не известен (другими словами, указатель `void` может представлять адрес любого объекта, но не его тип). Так как `void` не имеет типа, то к нему не применима операция разадресации (взятие содержимого) и адресная

арифметика, ведь неизвестно представление данных. Тем не менее, если мы работаем с указателем типа `void`, то нам доступны операции сравнения.

Переменная не может иметь типа `void`, этот тип определён только для указателей.

Пустые указатели нашли широкое применение при вызове функций, дальше мы увидим, как можно эффективно применить указатель `void`.

Арифметика указателей

Вы можете выполнить следующие операции на указателями:

- Прибавить целое число или вычесть целое число от указателя;
- Вычитать один указатель из другого;
- Сравнить указатели.

Когда вы вычитаете один указатель из другого, они должны иметь одинаковый базовый тип. Также вы можете сравнивать указатель с `null` указателем с помощью операторов сравнения (`==`, `!=`). И вы можете сравнивать указатель на объект с `void` указателем.

Чаще всего арифметика над указателями находит применение, когда мы работаем с массивами. Дальше мы рассмотрим подробнее эту тему, а пока простые примеры для иллюстрации:

```
#include <conio.h>
#include <stdio.h>

void main(int argc, char **argv) {
    int x = 100; // простая переменная x
    int y = 200; // простая переменная y
    int *p_x1 = &x; // указатель p_x1 на адрес x
    int *p_x2 = &x; // указатель p_x2 на адрес x
    int *p_y1;
    int temp;

    // выводим адреса указателей
    printf("address of p_x1 = %p\n", p_x1);
    printf("address of p_x2 = %p\n", p_x2);

    // сравниваем p_x1 и p_x2
    if (p_x1 == p_x2)
        printf("true\n");

    // прибавляем (2*4=8 байтов) к p_x1
    p_x1 = p_x1 + 2;

    // смотрим на результат
    printf("address of p_x1 after add value 2 = %p\n", p_x1);

    *p_y1 = &y;

    printf("address of p_y1 = %p\n", p_y1);

    // разность указателей
    temp = p_x2 - p_y1;

    printf("value of expressions p_x2 - p_y1 = %d\n", temp);
}
```

```

    // вычитаем обратно (2*4=8 байтов) от p_x1
    p_x1 = p_x1 - 2;

    // смотрим на результат и зрительно сравниваем с исходным
    printf("address of p_x1 after subtraction value 2 = %p\n", p_x1);

    _getch(); // у вас возможно просто getch();
}

```

Результат работы показан на Рис. 4-4.

Операция сложения вырабатывает адрес, который определяется следующим образом: (адрес в указателе) + (значение `int_выражения`)*`sizeof(<тип>)`), где `<тип>` это тип данных, на которые ссылается указатель. Операция вычитания вырабатывает адрес, который определяется так: (адрес в указателе) - (значение `int_выражения`)*`sizeof(<тип>)`. Адреса представлены в 16-ой системе счисления (A=10, B=11, C=12, D=13, E=14, F=15).

```

C:\Users\Arsen\Documents\Visual Studio 2015\Projects\c++\Pointers\Debug\Pointers
address of p_x1 = 001CFCB4
address of p_x2 = 001CFCB4
true
address of p_x1 after add value 2 = 001CFCBC
address of p_y1 = 001CFCA8
value of expressions p_x2 - p_y1 = 3
address of p_x1 after subtraction value 2 = 001CFCB4
-

```

Рис. 4-4 Результат арифметических операций с указателями

Почему в данном случае выражение `p_x2 - p_y1 = 3`? `p_x2 = 0037FCB8`, `p_y1 = 0037FCAC` => вычитаем `0037FCB8 - 0037FCAC = 0000000C` (12 байтов / 4 байта = 3, т.е. 3 объекта `int` по 4 байта).

Const указатели и указатели на const объекты

Когда вы определяете `const` (константный, неизменный) указатель, вы должны также инициализировать его, потому что вы не можете модифицировать указатель потом. Следующий пример иллюстрирует данную ситуацию:

```

int var; // переменная типа int
int *const ptr_var = &var; // константный указатель на int
*ptr_var = 123; // Ок: мы можем изменять данные переменной
++ptr_var; // Ошибка: мы не можем изменить указатель

```

Если вы указываете модификатор `const` после символа `*`, то неизменяемым будет именно указатель, т.е. нельзя изменить адрес куда он указывает в памяти. Если до символа `*`, то константными будут данные в переменной:

```

const int *ptr_var = &var; // константный указатель с const данными
*ptr_var = 123; // Ошибка: мы не можем изменять данные переменной

```

```
++ptr_var;    // Ок: а теперь мы можем изменить данные
```

Пример, когда неизменные и данные, и указатель:

```
const int *const ptr_var = &var;
```

Когда в будущем вы будете изучать C++ и объектно-ориентированное программирование (ООП) вы поймете, что **const** ваш лучший друг при передаче объектов в функции и методы, что не все методы класса должны модифицировать данные класса и т.п. А сейчас запомните одно: если необходимо, например, передать указатель в функцию или хранить указатели в массиве, но без возможности изменять данные, то указывайте **const**.

Итого

Прежде чем переходить к теме массивы и указатели, давайте вкратце вспомнить, что мы уже знаем, и рассмотрим еще наглядный пример.

- 1) Чтобы объявить указатель необходимо поставить знак `<*>` перед именем переменной: `char *ptr_value;`
- 2) Если тип указателя не совпадает с типом переменной, на адрес которой указывает, необходимо явное преобразование:

```
double x = 10.5;
char *p_x = &x; // Ошибка: не соответствие типа; нет явного
преобразования.
char *p_x = (char *)&x; // Ок: p_x указывает на первый байт x;
```

- 3) Не будем вдаваться в тонкости 32-разрядных и 64-разрядных платформ(процессоров), особенности компиляторов т.д. Просто запомните, что если вы компилируете свою программу под 32-разрядную платформу (а на лабораторных работах в Станкине вероятность этого 99.9%), то указатель занимает 4 байта;
- 4) `NULL` – константное выражение со значением 0, приведенное к `void *`. Можете использовать макрос `NULL` (определен в `stdlib.h`, `stdio.h` и других);
- 5) `void` – пустой указатель, пример:

```
void *ptr = NULL; // ptr ни на что не указывает
int x = 10;       // x - простая переменная типа int
ptr = &x;         // ptr теперь указывает на x
*((int *) ptr) = 50; // чтобы изменить значение x на 50, нужно сначала
привести к типу (int *), так как x типа int;
```

- 6) Вы можете выполнить следующие операции над указателями:
 - a. Прибавить целое число или вычесть целое число от указателя,
 - b. Вычитать один указатель из другого,
 - c. Сравнивать указатели;
- 7) `const` – модификатор, который поможет вам управлять доступом к данным:

```
const double x = 10; // неизменная простая переменная,
const int *p_y;      // нельзя модифицировать данные, на которые указывает
```

```
int *const z = &x;    // нельзя модифицировать сам указатель, т.е. адрес;
```

В качестве примера рассмотрим функцию `swap`, которая служит для обмена значениями двух ее аргументов. Если в обычном примере она принимает аргументы типа `int`, то сейчас сделаем, чтобы она принимала и другие типы данных. Здесь поможет `void` указатель.

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void swap(void *a, void *b, int size) {
    char temp;
    int i;
    if(a == NULL || b == NULL) // проверка на NULL
        return;
    for (i = 0; i < size; i++) {
        temp = *((char*)b + i);
        *((char*)b + i) = *((char*)a + i);
        *((char*)a + i) = temp;
    }
}

int main(int argc, char **argv) {
    float a = 10.f;
    float b = 20.f;
    double c = 555;
    double d = 777;
    unsigned long e = 211;
    unsigned long f = 311;

    printf("a = %.3f, b = %.3f\n", a, b);
    swap(&a, &b, sizeof(float));
    printf("a = %.3f, b = %.3f\n", a, b);

    printf("c = %.3f, d = %.3f\n", c, d);
    swap(&c, &d, sizeof(double));
    printf("c = %.3f, d = %.3f\n", c, d);

    printf("e = %ld, f = %ld \n", e, f);
    swap(&e, &f, sizeof(unsigned long));
    printf("e = %ld, f = %ld \n", e, f);

    _getch();
}
```

Результат выполнения функции:

```
C:\Users\Arsen\Documents\Visual Studio 2015\Projects\
a = 10.000, b = 20.000
a = 20.000, b = 10.000
c = 555.000, d = 777.000
c = 777.000, d = 555.000
e = 2, f = 3
e = 3, f = 2
```

Рис. 4-5 Результат работы функции swap

Функция swap принимает указатели на void, и дополнительный параметр типа int(unsigned int). Указатели a, b могут принимать адрес любого типа, но значение самих указателей нужно проверить. Параметр size нужен для определения количества байтов в типе данных, так как swap копирует данные побайтно.

Для понимания рассмотрим подробнее первый случай с float (размер 4 байта), цикл выполнится 4 раза, так как float занимает 4 байта в памяти. Temp – это временная переменная типа char (1 байт), для сохранения значения промежуточного байта. В выражении

```
Temp = *((char*)b + i)
```

Если i при первой итерации равна 0, а b (типа float) явно приводим к указателю типа char*, то после добавления 0 и разыменовывания указателя операндом * в temp записывается первый байт переменной b. Далее:

```
*((char*)b + i) = *((char*)a + i)
```

здесь в первый байт b записываем первый байт a.

```
*((char*)a + i) = temp
```

в первый байт a записываем первый байт b, ранее сохраненный в temp. Повторяем операцию еще 3 раза, так i меняет от 0 до 3, и каждый раз берем соответствующие номера байтов.

Индивидуальное задание №1.

Примечание: при решении задач необходимо применить указатели соответствующего типа и функции которые принимают аргументы указатели на данные. Внутри функций необходимо проверять что переданные указатели не нулевые (NULL).

1. Написать функцию, которая вычисляет объем цилиндра. Параметрами функции должны быть радиус и высота цилиндра. Тип данных unsigned long.

2. Написать функцию, которая возвращает значение максимального из трех дробных чисел, полученных в качестве аргумента. Тип данных float.

3. Написать функцию, которая сравнивает два натуральных числа и возвращает результат сравнения в виде одного из знаков: $>$, $<$ или $=$. Тип данных `double`.

4. Написать функцию, которая вычисляет сопротивление цепи, состоящей из двух резисторов. Параметрами функции являются величины сопротивлений и тип соединения (последовательное или параллельное). Функция должна проверять корректность параметров: если неверно указан тип соединения, то функция должна возвращать - 1. Тип данных `float`.

5. Написать функцию, которая вычисляет значение a^b . Числа a и b могут быть любыми дробными положительными числами. Тип данных для дробных чисел.

6. Написать функцию `Procent`, которая возвращает сколько процентов составляет первый полученный аргумент от второго полученного в качестве аргумента числа. Тип данных `double`.

7. Написать функцию "Факториал" и программу, использующую эту функцию для вывода таблицы факториалов. Тип данных `unsigned int`.

8. Написать функцию `Dohod`, которая вычисляет доход по вкладу. Исходными данными для функции являются: величина вклада, процентная ставка (годовых) и срок вклада (количество дней). Процентная ставка должна округляться до целых значений внутри функции и передаваться в вызывающий метод. Тип данных `float`.

9. Написать функцию `glasn`, которая возвращает 1, если символ, полученный функцией в качестве аргумента, является гласной буквой русского алфавита, и ноль — в противном случае. Тип данных `char`.

10. Напишите функцию, которая принимает два входных аргумента и передает вызывающему окружению два результата, первый из которых является произведением аргументов, а второй — их суммой. Поскольку из функции можно непосредственно вернуть только одно значение, придется вернуть второе значение при помощи второго параметра, являющегося указателем или ссылкой.

11. Напишите программу, которая сравнивает адреса памяти двух различных переменных стека и выводит переменные (вместе с адресами) на консоль в порядке роста их адресов.

12. Напишите функцию, которая принимает два входных аргумента и передает вызывающему окружению два результата, первый из которых является произведением аргументов, а второй — их суммой. Поскольку из функции можно непосредственно вернуть только одно значение, придется вернуть второе значение при помощи дополнительного параметра, являющегося указателем или ссылкой.

Указатели для работы с массивом

Возьмем обычный статический массив:

```
int massiv [size] = {3, 5, 8, 4, 1, 2, 9};  
int *ptr = massiv; // ptr указывает на адрес первого элемента массива
```

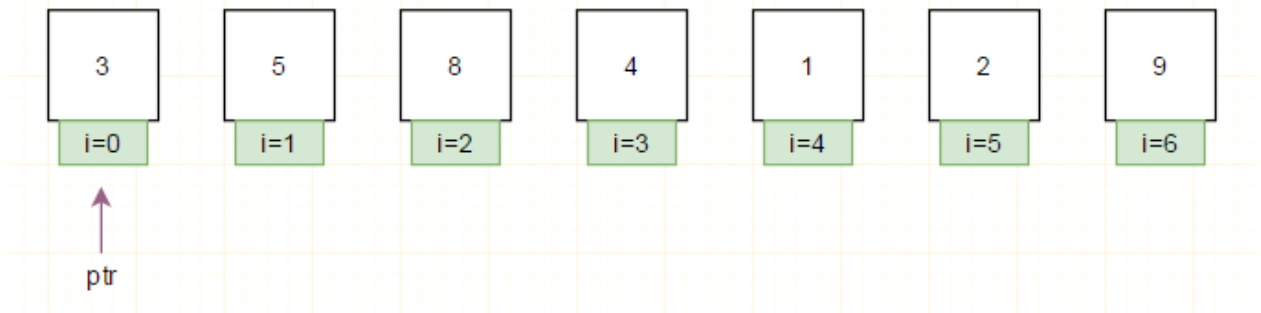


Рис. 4-6 Представление массива в памяти

Программа для обхода массива с помощью указателя:

```
#include <stdio.h>  
#include <conio.h>  
  
int main(int argc, char **argv)  
{  
    // объявим и выведем на печать массив  
    int massiv[7] = { 3, 5, 8, 4, 1, 2, 9 };  
    int *ptr = massiv; // указатель установлен на начало массива  
  
    for (int i = 0; i < 7; i++)  
    {  
        printf("array[%i] = %i\n", i, massiv[i]);  
    }  
    // с помощью указателя  
    // вывод значения первого элемента массива  
    printf("prt = %p\n", ptr);  
    printf("*prt = %i\n", *ptr);  
  
    int i = 0;  
    // вывод всех значений  
    while (i < 7)  
    {  
        printf("array[%i] = %i\n", i, *ptr++); //указатель увеличивается на 1  
        i++;  
    }  
    _getch();  
}
```

Результат представлен на рисунке ниже:

```
C:\Users\Arsen\Documents\Visual Studio 2015\Projects\c++\Pointers\De
array[0] = 3
array[1] = 5
array[2] = 8
array[3] = 4
array[4] = 1
array[5] = 2
array[6] = 9
prt = 003DFAAC
*prt = 3
array[0] = 3
array[1] = 5
array[2] = 8
array[3] = 4
array[4] = 1
array[5] = 2
array[6] = 9
```

Рис. 4-7 Результат выполнения программы

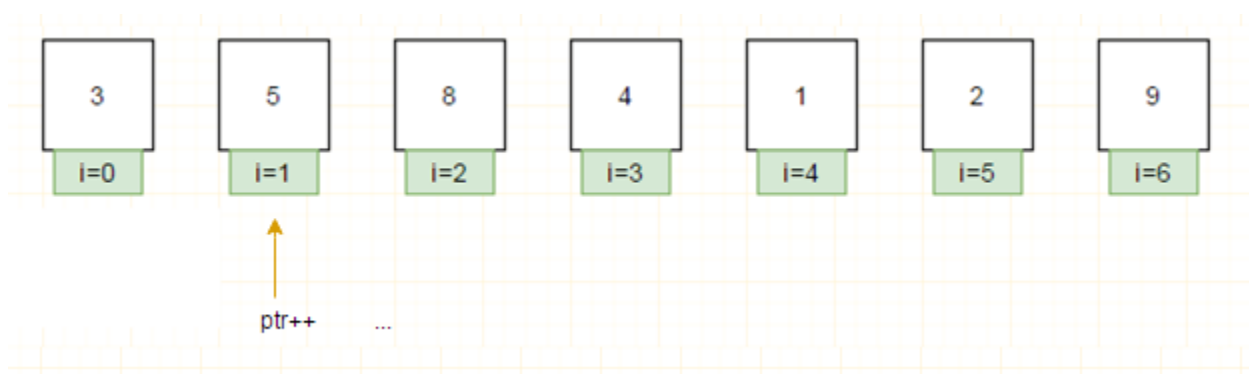


Рис. 4-8 Представление массива в памяти

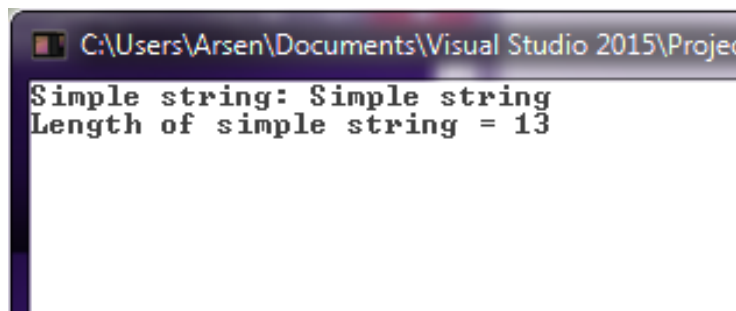
Как вы знаете, операция инкрементации ($\text{ptr}++$) аналогична $\text{ptr} = \text{ptr} + 1$; 1 – блок памяти, который равняется количеству байтов в типе данных, т.е. у нас указатель типа `int` (4 байта), значит $\text{ptr} = \text{ptr} + \text{int}(4 \text{ байта})$, поэтому, инкрементируя указатель по массиву типа `int`, мы передвигаемся по его ячейкам.

Прежде чем продолжить, изучите и запомните следующие стандартные функции для работы с массивами:

Стандартные функции

1. `size_t strlen(char *str)` – возвращает длину строки `str`;

```
// массив типа char, т.е. строка, указатель *string указывает на первый
элемент
const char *string = "Simple string\0";
printf("Simple string: ");
puts(string); // выводим строку string
printf("Length of simple string = %i\n", strlen(string)); // длина строки
string
```

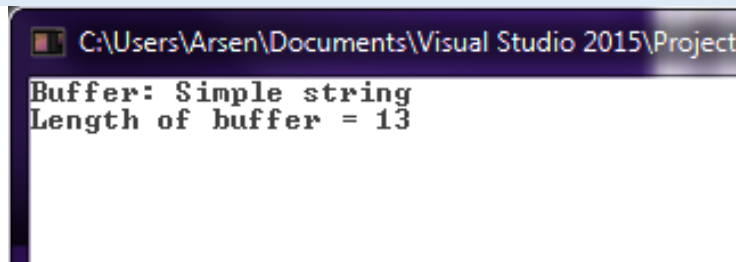


```
C:\Users\Arsen\Documents\Visual Studio 2015\Project
Simple string: Simple string
Length of simple string = 13
```

Рис. 4-9 Результат работы с функцией strlen

2. `char *strcpy(char *dest, const char *src)` – копирует из src в dest;

```
// простой буфер
char buffer[100];
strcpy(buffer, string); // копируем строку string в buffer
printf("Buffer: ");
puts(buffer); // выводим buffer
printf("Length of buffer = %i", strlen(buffer)); // проверяем длину
буфера
```



```
C:\Users\Arsen\Documents\Visual Studio 2015\Project
Buffer: Simple string
Length of buffer = 13
```

Рис. 4-10 Результат работы с функцией strcpy

3. `int strcmp(const char *lhs, const char *rhs)` – сравнивает две строки в лексикографическом порядке. Возвращает: 1) 0, если строки равны, 2) меньше нуля, если lhs < rhs, 3) больше нуля, если lhs > rhs;

```
char fruit[] = "apple"; // загаданный фрукт
char answer[80]; // строка-ответ
do
{
    puts("Guess my favorite fruit?");
    gets(answer);
} while(strcmp(fruit, answer) != 0); // пока фрукт не отгадан, цикл будет
работать
puts("Correct answer!\n");
```

```
C:\Users\Arsen\Documents\Visual Studio 2015\Proje
Guess my favorite fruit?
orange
Guess my favorite fruit?
banana
Guess my favorite fruit?
pear
Guess my favorite fruit?
apple
Correct answer!
```

Рис. 4-11 Результат работы с функцией strcmp

4. `void qsort(const void *ptr, size_t count, size_t size, int (*comp)(const void *, const void *))` – стандартная функция сортировки по возрастанию, где **ptr** – указывает на сортируемый массив, **count** - число элементов в массиве, **size** - размер каждого элемента массива в байтах, **comp** – функция используется для сравнения объектов. **Comp** возвращает отрицательное значение, если первый аргумент меньше второго, положительно целое значение, если первый аргумент больше второго и нуль, если аргументы равны;

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
// функция сравнения
int compare_ints(const void* a, const void* b)
{
    const int *arg1 = (int*)a;
    const int *arg2 = (int*)b;

    return *arg1 - *arg2;
}

int main(int argc, char **argv)
{
    int i;
    int ints[] = { -2, 99, 0, -743, 2, 3, 4 };
    // size равен 4 * 7 / 4 = 7
    int size = sizeof ints / sizeof *ints;

    qsort(ints, size, sizeof(int), compare_ints);

    for (i = 0; i < size; i++)
        printf("%d ", ints[i]);

    printf("\n");

    _getch();
    return EXIT_SUCCESS;
}
```

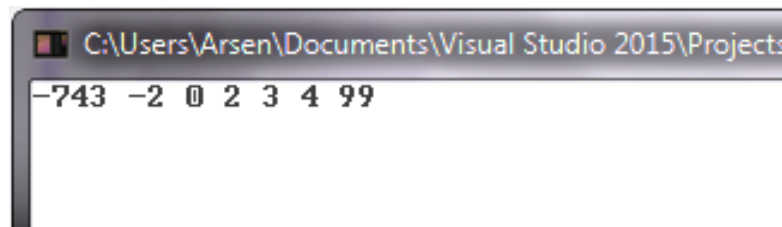


Рис. 4-12 Результат работы с функцией qsort

5. **void* malloc(size_t size)** - выделяет **size** байт в хранилище(куче, оперативной памяти). Инициализация выделенной памяти не производится. В случае успешного выделения памяти, производится возврат указателя на наименьший (первый) байт. Блок памяти выравнивается, чтобы подходить для любого типа объектов. Если значение **size** равно нулю, то поведение функции определяется реализацией (может быть возвращен либо указатель на **null**, либо указатель на не **null**, который нельзя использовать для доступа к хранилищу);
6. **void free(void* ptr)** - Освобождает пространство, ранее выделенное **malloc()**, **calloc()** или **realloc()**. Если **ptr** является нулевым указателем, то функция ничего не делает. Поведение не определено, если **ptr** не соответствует указателю, возвращенным ранее **malloc()**, **calloc()** или **realloc()**. Кроме того, поведение не определено, если область памяти, на которую ссылается **ptr**, уже была освобождена, то есть **free()** или **realloc()** уже был вызван с **ptr** в качестве аргумента.

```
// Выделение памяти, достаточной для массива из 4 int
int* p1 = (int*)malloc(4 * sizeof(int));
// Тоже самое, но с прямым указанием типа
int* p2 = (int*)malloc(sizeof(int[4]));
// Тоже самое, но без повторного указания имени типа
int* p3 = (int*)malloc(4 * sizeof *p3);

if (p1) {
    for (int n = 0; n<4; ++n) // Заполнение массива
        p1[n] = n*n;
    for (int n = 0; n<4; ++n) // Вывод его содержимого
        printf("p1[%i] == %i\n", n, p1[n]);
}
// освобождаем память!
free(p1);
free(p2);
free(p3);
```

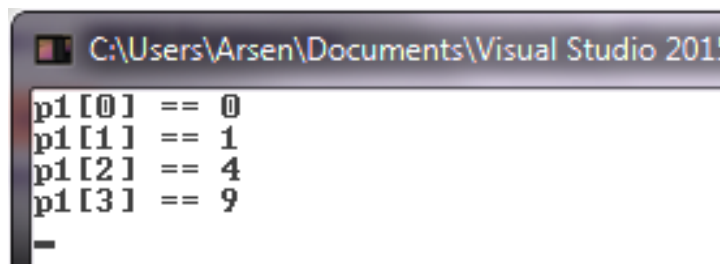


Рис. 4-13 Результат работы с функциями malloc и free

Индивидуальное задание №2. Одномерные массивы и указатели.

Примечание: Для решение использовать указатели и динамическое выделение памяти

1. Объявите указатель на массив типа `double` и предложите пользователю выбрать его размер. Далее напишите четыре функции: первая должна выделить память для массива, вторая — заполнить ячейки данными, третья — показать данные на экран, четвертая — освободить занимаемую память.
2. Написать программу, в которой объявлен указатель на массив типа `int` и выделенна память для 12-ти элементов. Необходимо написать функцию, которая поменяет значения четных и нечетных ячеек массива.
3. Написать программу, создающую массив из 10 случайных целых чисел введенных пользователем из отрезка $[-50;50]$. Вывести на экран весь массив и на отдельной строке — значение минимального элемента массива. Для обхода массива использовать указатели (запрещено обращаться к элементам массива по индексам).
4. Написать программу, которая (без использования библиотечных функций для обработки строк), копировала бы строку введенную пользователем с клавиатуры в новую (максимальная длина строки — 80 символов). При этом в процессе копирования должны отбрасываться все незначащие пробелы в начале и конце строки, а также несколько подряд идущих пробелов должны заменяться на один. Вывести исходную и новую строки на экран. Для обхода строк использовать указатели.
5. Написать программу, которая для введенной с клавиатуры строки (максимальная длина строки — 80 символов) сообщает, какая цифра в ней встречается чаще всего, либо сообщает, что цифры в строке совсем отсутствуют. Для обхода строк использовать указатели.
6. Написать программу, которая для введенной пользователем с клавиатуры строки (максимальная длина строки — 80 символов) программа должна определить, корректно ли расставлены скобки (круглые, фигурные, квадратные) или нет. Перемешивание скобок (пример: «{[]}») считается некорректным вариантом.
7. Написать программу, которая формирует массив натуральных чисел заданного пользователем размера. Необходимо написать функцию, которая заполняет переданный ей массив значениями функции $y=x^2+2x-3$ из заданного диапазона значений. Для помощи можно использовать <http://www.yotx.ru>.
8. Написать программу, которая формирует массив целых чисел заданного пользователем размера. Необходимо написать функцию, которая заполняет переданный ей массив значениями функции $y=-2x^2+8x-3$

находящимися рядом с экстремумом (максимальным значением). Для помощи можно использовать <http://www.yotx.ru>.

9. Написать программу, которая формирует массив натуральных чисел заданного пользователем размера. Необходимо написать функцию, которая заполняет переданный ей массив значениями функции $y=x^2+4x-3$ находящимися рядом с экстремумом (минимальным значением). Для помощи можно использовать <http://www.yotx.ru>.
10. Написать программу, которая во введенной пользователем строке (максимальная длина строки — 80 символов) заменяет пробелы на указанные пользователем символ ('-', '+', '_') и выводит результат. Необходимо написать функцию, которая запрашивает значение символа для замены и модифицирует строку.
11. Написать программу, которая хранит введенную строку и имеет возможность ее дополнять новыми символами. Для хранения строки необходимо использовать динамический массив, размер которого должен увеличиваться по мере добавления новых данных.
12. Написать программу, которая сначала переворачивает слова (разделенные пробелами) в введенной пользователем строке, а потом меняет их местами так чтобы первое слово стало последним а последнее первым. Таким образом строку можно будет прочитать справа на лево. Пример вывода строк:
Первое слово Мир -> еовреП оволс риМ -> риМ оволс еовреП

Динамическое выделение памяти для двумерных массивов

Пусть требуется разместить в динамической памяти матрицу, содержащую n строк и m столбцов. Двумерная матрица будет располагаться в оперативной памяти в форме ленты, состоящей из элементов строк. При этом индекс любого элемента двумерной матрицы можно получить по формуле

$$\text{index} = i * m + j;$$

где i - номер текущей строки; j - номер текущего столбца.

Рассмотрим матрицу 3×4 (см. рис.)

		j=2			
		0	1	2	3
i=1		4	5	6	7
		8	9	10	11

Рис. 4-14 Матрица значений 3×4

Индекс выделенного элемента определится как

index = 1*4+2=6

Объем памяти, требуемый для размещения двумерного массива, определится как

Однако поскольку при таком объявлении компилятору явно не указывается количество элементов в строке и столбце двумерного массива, традиционное обращение к элементу путем указания индекса строки и индекса столбца является некорректным:

a[i][j] - некорректно.

Правильное обращение к элементу с использованием указателя будет выглядеть как

***(p+i*m+j),**

где p - указатель на массив, m - количество столбцов, i - индекс строки, j - индекс столбца.

Пример Ввод и вывод значений динамического двумерного массива

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main()
{
    int *a; // указатель на массив
    int i, j, n, m;
    system("chcp 1251");
    system("cls");
    printf("Введите количество строк: ");
    scanf("%d", &n);
    printf("Введите количество столбцов: ");
    scanf("%d", &m);
    // Выделение памяти
    a = (int*) malloc(n*m*sizeof(int));
    // Ввод элементов массива
    for(i=0; i<n; i++) // цикл по строкам
    {
        for(j=0; j<m; j++) // цикл по столбцам
        {
            printf("a[%d][%d] = ", i, j);
            scanf("%d", (a+i*m+j));
        }
    }
    // Вывод элементов массива
    for(i=0; i<n; i++) // цикл по строкам
    {
        for(j=0; j<m; j++) // цикл по столбцам
        {
            printf("%5d ", *(a+i*m+j)); // 5 знакомест под элемент массива
        }
        printf("\n");
    }
    free(a);
    getchar(); getchar();
    return 0;
}
```

Результат выполнения

```
c:\Projects\unoname\Debug\unoname.exe
Введите количество строк: 3
Введите количество столбцов: 4
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[0][3] = 4
a[1][0] = 5
a[1][1] = 6
a[1][2] = 7
a[1][3] = 8
a[2][0] = 9
a[2][1] = 10
a[2][2] = 11
a[2][3] = 12
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

Рис. 4-15 Результат выполнения программы

Возможен также другой способ динамического выделения памяти под двумерный массив - с использованием массива указателей. Для этого необходимо:

- выделить блок оперативной памяти под массив указателей;
- выделить блоки оперативной памяти под одномерные массивы, представляющие собой строки искомой матрицы;
- записать адреса строк в массив указателей.

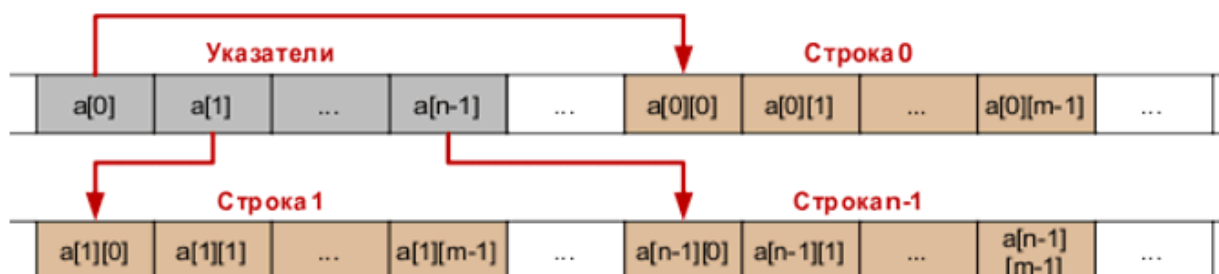


Рис. 4-16 Размещение элементов динамического двумерного массива в памяти компьютера

Графически такой способ выделения памяти можно представить следующим образом.

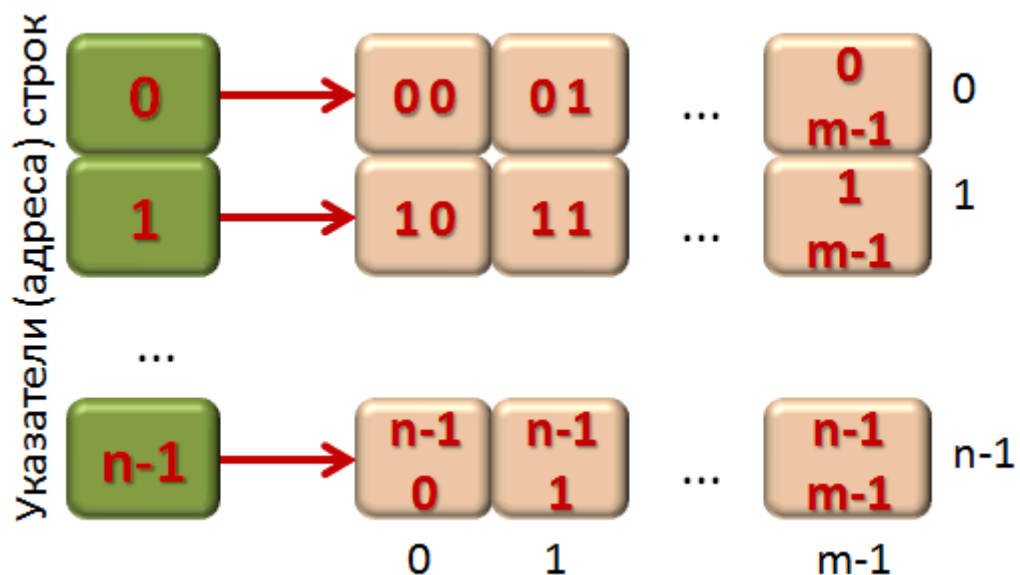


Рис. 4-17 Графическое представление динамического двумерного массива

При таком способе выделения памяти компилятору явно указано количество строк и количество столбцов в массиве.

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main()
{
    int **a; // указатель на указатель на
    int i, j, n, m;
    system("chcp 1251");
    system("cls");
    printf("Введите количество строк: ");
    scanf("%d", &n);
    printf("Введите количество столбцов: ");
    scanf("%d", &m);
    // Выделение памяти под указатели на строки
    a = (int**)malloc(n*sizeof(int*));
    // Ввод элементов массива
    for(i=0; i<n; i++) // цикл по строкам
    {
        // Выделение памяти под хранение строк
        a[i] = (int*)malloc(m*sizeof(int));
        for(j=0; j<m; j++) // цикл по столбцам
        {
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }
    // Вывод элементов массива
    for(i=0; i<n; i++) // цикл по строкам
    {
        for(j=0; j<m; j++) // цикл по столбцам
        {
            printf("%5d ", a[i][j]); // 5 знакомест под элемент массива
        }
        printf("\n");
    }
}
```

```

    free(a[i]);    // освобождение памяти под строку
}
free(a);
getchar();
return 0;
}

```

Результат выполнения программы аналогичен предыдущему случаю.

С помощью динамического выделения памяти под указатели строк можно размещать свободные массивы. Свободным называется двухмерный массив (матрица), размер строк которого может быть различным. Преимущество использования свободного массива заключается в том, что не требуется отводить память компьютера с запасом для размещения строки максимально возможной длины. Фактически свободный массив представляет собой одномерный массив указателей на одномерные массивы данных.

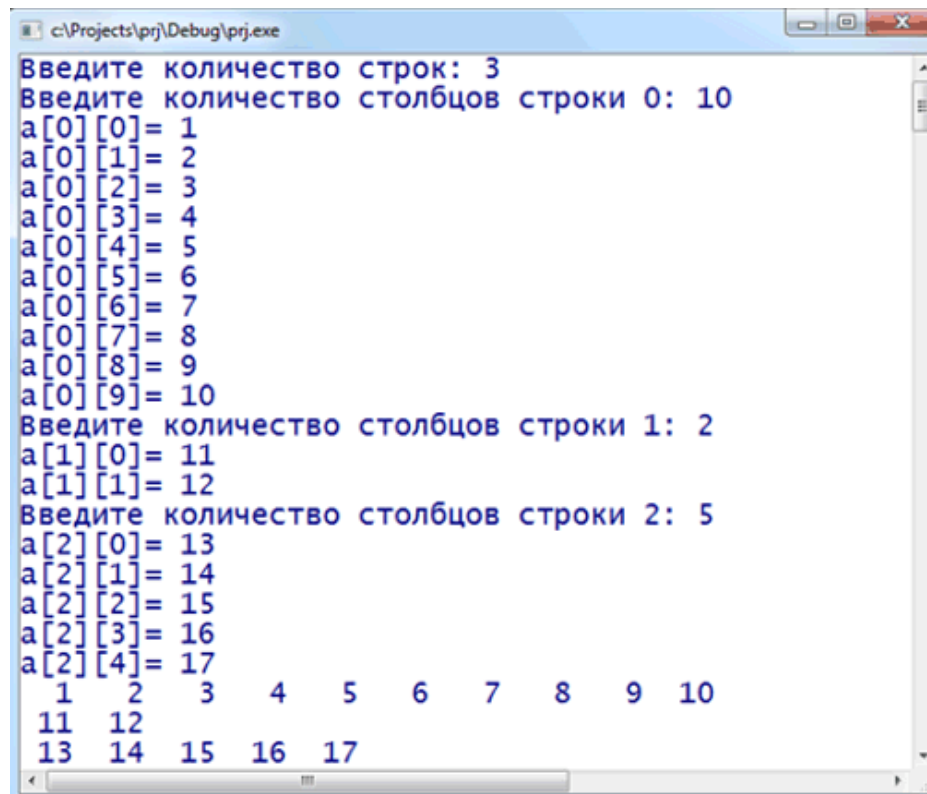
Для размещения в оперативной памяти матрицы со строками разной длины необходимо ввести дополнительный массив *m*, в котором будут храниться размеры строк.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int **a; // указатель на указатель
    int i, j, n, *m;
    system("chcp 1251");
    system("cls");
    printf("Введите количество строк: ");
    scanf("%d", &n);
    a = (int**)malloc(n*sizeof(int*));
    m = (int*)malloc(n*sizeof(int)); // массив кол-ва элементов строк
    // Ввод элементов массива
    for(i = 0; i<n; i++)
    {
        printf("Введите количество столбцов строки %d: ", i);
        scanf("%d", &m[i]);
        a[i] = (int*)malloc(m[i]*sizeof(int));
        for(j = 0; j<m[i]; j++)
        {
            printf("a[%d][%d]= ", i, j);
            scanf("%d", &a[i][j]);
        }
    }
    // Вывод элементов массива
    for(i = 0; i<n; i++)
    {
        for(j = 0; j<m[i]; j++) {
            printf("%3d ", a[i][j]);
        }
        printf("\n");
    }
    getchar();
    return 0;
}

```

Результат выполнения



```
c:\Projects\prj\Debug\prj.exe
Введите количество строк: 3
Введите количество столбцов строки 0: 10
a[0][0]= 1
a[0][1]= 2
a[0][2]= 3
a[0][3]= 4
a[0][4]= 5
a[0][5]= 6
a[0][6]= 7
a[0][7]= 8
a[0][8]= 9
a[0][9]= 10
Введите количество столбцов строки 1: 2
a[1][0]= 11
a[1][1]= 12
Введите количество столбцов строки 2: 5
a[2][0]= 13
a[2][1]= 14
a[2][2]= 15
a[2][3]= 16
a[2][4]= 17
  1   2   3   4   5   6   7   8   9  10
11  12
13  14  15  16  17
```

Рис. 4-18 Заполнение и вывод свободного двумерного массива

Индивидуальное задание №3. Двумерные массивы и указатели.

Примечание: Для решение использовать указатели и динамическое выделение памяти

1. Объявить и заполнить двумерный динамический массив случайными числами от 10 до 50. Показать его на экран. Для заполнения и показа на экран написать отдельные функции (функции должны принимать три параметра — указатель на динамический массив, количество строк, количество столбцов). Количество строк и столбцов выбирает пользователь.
2. Заполнить двумерный массив случайными числами от 10 до 100. Количество строк и столбцов должен задавать пользователь. Посчитать сумму элементов отдельно в каждой строке и определить номер строки, в которой эта сумма максимальна.
3. Объявить двумерный массив и заполнить его построчно с клавиатуры. Количество строк и столбцов должен задавать пользователь. После заполнения показать заполненную матрицу на экран и посчитать сумму элементов отдельно в каждом столбце и каждой строке.
4. Объявить двумерный массив, заполнить целыми числами и показать на экран. Количество строк и столбцов должен задавать пользователь. После заполнения произвести сортировку строк по возрастанию.

-
5. Объявить двумерный массив, заполнить целыми числами и показать на экран. Количество строк и столбцов должен задавать пользователь. После заполнения произвести сортировку столбцов по возрастанию.
 6. Объявить двумерный массив и заполнить его построчно с клавиатуры. Количество строк и столбцов должен задавать пользователь. После заполнения показать заполненную матрицу на экран и посчитать сумму элементов для двух диагоналей матрицы чисел.
 7. Объявить свободный двумерный массив (двумерный массив со строками различной длины) и заполнить его построчно с клавиатуры. Для каждого столбцов значения которых есть в каждой строке посчитать среднее арифметическое значение.
 8. Объявить свободный двумерный массив (двумерный массив со строками различной длины) и заполнить его построчно с клавиатуры. Для столбцов значения которых есть в каждой строке посчитать среднее арифметическое значение.
 9. Объявить свободный двумерный массив (двумерный массив со строками различной длины) и заполнить его построчно с клавиатуры. Для всех элементов из столбцов, значения которых есть в каждой строке, посчитать среднее арифметическое значение.
 10. Объявить двумерный массив и заполнить его построчно с клавиатуры. Необходимо написать функцию, которая изменяет значения элементов массива – для элементов значения которых меньше указанного устанавливает значение 0.
 11. Объявить двумерный символьных массив и заполнить его построчно с клавиатуры. Необходимо написать функцию, которая изменяет значения элементов массива – для элементов содержащих гласные буквы устанавливает значение символа нижнего подчеркивания ('_').
 12. Заполнить двумерный массив случайными числами от 10 до 100. Количество строк и столбцов должен задавать пользователь. Вывести на консоль массив четырех представлениях:
 - a. с поворотом значений элементов на 0°
 - b. с поворотом значений элементов на 90°
 - c. с поворотом значений элементов на 180°
 - d. с поворотом значений элементов на 270°