Лабораторная работа №6

Указатели.

Цель работы: освоить принципы работы с указателями в Си, выполнить упражнения по вариантам

Краткие теоретические сведения

Указатель – особый вид переменной, которая хранит *адрес* элемента памяти, где может быть записано *значение другой* переменной.

Определение указателя:

имя_типа *имя_переменной;

Пример:

```
int var, *point;
```

Здесь *var* – целочисленная переменная, point указатель на целый тип. Символ * - звездочка определяет тип “указатель”.

Существует операция, неразрывно связанная с указателями. Это унарная операция – операция взятия *адреса* **&**. Результат операции & является адрес переменной в памяти. Результат имеет тип «указатель» на тип переменной. Операция **&** может использоваться практически со всеми типами данных, кроме *констант* и *битовых полей*.

Пример:

```
int var, *point;
point = &variable;
```

Указатели часто используются для обмена данными с функциями. Функции можно передать столько аргументов, сколько требуется, а вернуть с помощью return можно только одно значение. Когда возникает необходимость вернуть в вызывающую функцию более одного значения, то используются указатели.

Доступа по указателю (*разадресации*) (***point**). Результат операции – содержимое ячейки памяти, на которую указывает ***point***.

Пример 1:

```
#include<stdio.h>
void main(void)
{   int *p, a=18;
    p=&a;
    printf(“a = %d = %d\n”, a, *p);
    printf(“адрес a = %d = %d\n”, &a, p);
    *p+=8;   // a=26
    printf(“a = %d = %d\n”, a, *p);
}
```

Операция *присваивания* (если указатели разных типов, необходимо применить операцию приведения);

Пример:

```
int a=10, *p, *t;
p=&a;
t=p
```

Операция *увеличения* (*уменьшения*) указателя (***point+i; point-i;*** i- значение целочисленного типа); Результат операции (***point+i***) – "указатель", определяющий адрес i-го элемента после данного, а (***point+i***) – на i-ый элемент перед данным.

Операции *сложного присваивания* (***point+=i, point-=i***).

Операция *инкремента* (*декремента*): ***point++; point--; ++point; --point***. Указатель будет смещаться (увеличиваться или уменьшаться в зависимости от операции) на один элемент. Фактически указатель (адрес) измениться на количество байт, занимаемых этим элементом в памяти.

Для операций увеличения (уменьшении) указателя справедливо, если объявлен:

type1 *pointer;

где *type1* – некоторый тип, то операция pointer=pointer+i, где i- значение целочисленного типа, изменит pointer на ***sizeof****(type1)*i байт*.

Пример:

```
int a, *pi=&a;
float f, *pf=&f;
pi++;   //смещение на 2 байта
pf++;   //смещение на 4 байта
```

Операция *индексирования*: ***point[i]***. Эта операция полностью аналогична выражению ****(point+i)***, т.е. извлекается из памяти и используется в выражении значение i-го элемента массива, адрес которого присвоен указателю ***point***.

Операция вычитания: ***point1-point2*** (операция имеет смысл только в том случае, если ***point1*** и ***point2*** указатели на один и тот же набор данных). Результат имеет тип ***int*** и равен количеству элементов, которые можно расположить между ячейками памяти, на которые указывают ***point1*** и ***point2.***

Операции *отношения*:

point1==point1;
point1>=point2; point1>point2;
point1<=point2; point1<point2;
point1!=point1;

Также любой указатель (адрес) может быть проверен на равенство (==) или неравенство (!=) со специальным значением **NULL,** т.е. производиться сравнение с 0.

Связь между указателями и массивами

Имя массива – это ***указатель***, равный адресу начального массива (1-го байта 1-го элемента массива).

При объявлении **int** m[10]; одновременно с выделением памяти для десяти элементов типа **int**, определяется значение указателя m, значение m равно адресу элемента ***m[0]***. Значение ***m*** изменить нельзя, т.к. оно является константой. Индексные выражения и адресные эквивалентны: m[i] ⇔ *(m+i).

Для двухмерных массивов имя является указателем-константой на массив указателей-констант. Элементами массива указателей являются указатели-константы на начало каждой из строк массива.

Пример 2: *Используя указатели ввести одномерный массив вещественных чисел из n элементов, подсчитать сумму элементов, найти минимальный и максимальный элемент.*

```
#include <stdio.h>
#include <conio.h>
void main ()
{
        int i, kol=0, n;
        float b[30], s=0, max, min;
        do{
                printf("Vvedite kol-vo elementov massiva (<30)\n");
                scanf("%d",&n);
        } while (n>=30);
        for (i=0;i<n;i++)
        {
                printf("Vvedite element [%d]\n", i+1);
                scanf("%f",b+i);
        }
        for (i=0;i<n;i++)
                s+=*(b+i);
        printf("Summa elementov matrici = %.2f\n", s);

        max=*b;
        for (i=0;i<n;i++)
                if (*(b+i)>max)
                        max=*(b+i);
        printf("Max element matrici = %.2f\n", max);

        min=*b;
        for (i=0;i<n;i++)
                if (*(b+i)<min)
                        min=*(b+i);
        printf("Min element matrici = %.2f\n", min);

}
```

Динамическое выделение памяти. Одним из способов хранения информации является использование системы динамического выделения памяти языка Си. При этом память выделяется из свободной области памяти по мере надобности и возвращается назад, т.е. освобождается, когда необходимость в ней исчезла. Область свободной памяти, доступной для выделения, находится между областью памяти, где размещается программа, и стеком. Эта область называется *кучей* или *хипом* (от англ. heap – куча).

Так как память выделяется по мере необходимости и освобождается, когда её использование завершилось, то можно использовать ту же самую память в другой момент времени и для других целей в другой части программы. Динамическое выделение памяти даёт возможность создания динамических структур данных: списков, деревьев и др.

Ядром динамического выделения памяти в Си являются функции, объявленные в стандартной библиотеке в заголовочном файле **stdlib.h** – **malloc()**, **calloc()**, **realloc()** и **free()**.

Рассмотрим каждую функцию в отдельности.

void *malloc(unsigned size)

Функция **malloc** выделяет из хипа область памяти размером *size* байт. В случае успеха, **malloc** возвращает указатель на начало выделенного блока памяти. Если для выделения блока в хипе не хватает памяти, возвращается **NULL**. Содержимое памяти блока оставляется неизменным. Если размер аргумента равен нулю, **malloc** возвращает **NULL.**

Пример 3: *Программа выделяет динамическую память под массив из n элементов, запрашивает массив с клавиатуры и выводит его на экран.*

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
        int *ptr, i, n;
        do{
                printf("Vvedite kol-vo elementov massiva (<30)\n");
                scanf("%d",&n);
        } while (n>=30);

        if(!(ptr=(int*)malloc(n*sizeof(int))))   //Необходимо всегда
   {    puts("Not enough memory");          // проверять выделилась
        getch();                            // ли память
        return;
    }
     //ptr указывает на массив из n элементов
        for (i=0;i<n;i++)
        {
```

```
            printf("Vvedite element [%d]\n", i+1);
            scanf("%d",ptr+i);
        }
        printf("\nMassiv: \n", i+1);
        for (i=0;i<n;i++)
            printf("%d ", *(ptr+i));
}
```

void *calloc(unsigned num, unsigned size)

Функция **calloc** выделяет блок памяти и возвращает указатель на первый байт блока. Размер выделяемой памяти равен величине *num*size*, т.е. функция выделяет память, необходимую для хранения массива из num элементов по size байт каждый. В случае нехватки памяти для удовлетворения запроса **calloc** возвращает **NULL**. Выделяемая память инициализируется *нулями*.

Пример:

```
void main()
   { int *ptr;
.....................
      if (!(ptr=(int*)calloc(5,sizeof(int))))      // ptr указывает
         {puts("Not enough memory");               //на массив из 5
           getch(); return;                         //элементов, заполненный нулями
         }
.....................
   }
```

void *realloc(void *ptr, unsigned size)

Эта функция изменяет *размер* динамически выделяемой области памяти, на которую указывает **ptr*, на *size* (новый размер). Если указатель не является значением, которое ранее было определено функциями **malloc**, **calloc** или **realloc**, то поведение функции не определено. Это же справедливо, если *ptr* указывает на область памяти, ранее освобождённую функцией **free**. Значение *size* является абсолютным, а не относительным, т.е. задаёт новый размер блока, а не приращение старого. Если *size* больше, чем размер ранее существовавшего блока, то новое, неинициализированное, пространство памяти будет выделено в конце блока и предыдущее содержимое пространства сохраняется. Если **realloc** не может выделить память требуемого размера, то возвращаемое значение равно **NULL** и содержимое пространства, на которое указывает *ptr*, остаётся нетронутым. Если *ptr* – не **NULL**, а значение *size* равно нулю, то функция **realloc** действует как **free**.

Из вышесказанного следует сделать вывод о том, что когда бы размер блока памяти ни подвергся изменению под воздействием функции **realloc**,

новое пространство может начинаться с адреса, отличного от исходного, даже если **realloc** "*усекает*" память. Следовательно, если используется **realloc**, возникает необходимость следить за указателями на изменяемое пространство. Например, если вы создаёте связный список и выделяете при помощи **realloc** больший или меньший участок памяти для цепочки, то может оказаться, что цепочка будет перемещена. В этом случае указатели элементов будут адресоваться к участкам памяти, ранее занимаемым звеньями цепочки, а не в место их теперешнего расположения. Всегда следует использовать **realloc** так, как показано ниже:

if(p2 = = **realloc**(p1,new_size)) p1=p2;

Действуя подобным образом, вам никогда не придётся заботиться, выделялось ли для объекта новое пространство, т.к. p1 обновляется при каждом новом вызове функции, которая указывает на область памяти (возможно, новую).

Пример:

```
void main()
{
  int *ptr, tmp;
.....................
  if(!(ptr=(int*)calloc(5,sizeof(int))))          //ptr указывает на
   { puts("Not enough memory");                   // массив из 5 элементов,
     getch(); return;                              // заполненный нулями
    }
.....................
 if (tmp=realloc(ptr,10*sizeof(int))) ptr=tmp;    // ptr
   else
      {
         puts("Not enough memory");               //указывает на массив
         getch(); return;                          //из 10 элементов
      }
.....................
}
```

void free(void *ptr)

Функция *освобождает* область памяти, ранее выделенную при помощи функций **malloc**, **calloc** или **realloc**, на которую указывает ptr. Если ptr – **NULL**, то **free** ничего не выполняет. Если ptr не является указателем, проинициализированным ранее одной из функций выделения памяти, то поведение функции не определено. Заметим, что функция **free** не располагает средствами передачи ошибки, возможно возникающей при её выполнении, равно как возвращаемым значением.

Пример:

```
void main()
   { int *ptr;
.....
      if (!(ptr=(int*)calloc(5,sizeof(int))))      // ptr указывает
         { puts("Not enough memory");              // на массив из 5
           getch(); return;                        //элементов, заполненный нулями
         }
.....
     free(ptr);                    //ptr указывает на область памяти, ранее
                                   //занимаемую массивом
.....
        }
```

В вызовах функций можно использовать переменные типа **unsigned int** либо выражения, имеющие результатом тип **unsigned int.** Это соответствие также налагает ограничение на выделяемую память размером 64 Кбайт.

Пример 4: *Программа сортирует массив из 10 элементов по возрастанию методом пузырька.*

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

     void main()
     {
          int *b, i, j, n, buf;
          printf("Vvedite kol-vo elementov \n");
          scanf("%d",&n);
          b=new int[n];
          for (i=0;i<n;i++)
          {
               printf("Vvedite element [%d]\n", i+1);
               scanf("%d",b+i);
          }

          printf("\nIshodni massiv: \n", i+1);
          for (i=0;i<n;i++)
               printf("%d ", *(b+i));

          //сортировка массива методом пузырька
          for (i=0; i<n; i++)
               for (j=0; j<n-i-1; j++)
                    if (*(b+j)>*(b+j+1))
                    {
                         buf=*(b+j);
                         *(b+j)=*(b+j+1);
                         *(b+j+1)=buf;
                    }
          printf("\nOtsortirovani massiv: \n", i+1);
          for (i=0;i<n;i++)
               printf("%d ", *(b+i));
          }
```

Практическая часть

Упражнение 1

Написать программу в соответствии с вариантом, выделив под массив динамически память. Обращаться к элементам массива необходимо используя указатель.

1. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
- сумму отрицательных элементов массива;
- произведение элементов массива, расположенных между максимальным и минимальным элементами.

2. В одномерном массиве, состоящем из n целых элементов, вычислить:
- произведение элементов массива с четными номерами;
- сумму элементов массива, расположенных между первым и последним нулевыми элементами.

3. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
- максимальный элемент массива;
- сумму элементов массива, расположенных до последнего положительного элемента.

4. В одномерном массиве, состоящем из n целых элементов, вычислить:
- номер максимального элемента массива;
- произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

5. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
- максимальный по модулю элемент массива;
- сумму элементов массива, расположенных между первым и вторым положительными элементами.

6. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
- номер максимального по модулю элемента массива;
- сумму элементов массива, расположенных после первого положительного элемента.

7. В одномерном массиве, состоящем из вещественных элементов, вычислить:
- количество элементов массива, больших C;
- произведение элементов массива, расположенных после максимального по модулю элемента.

8. В одномерном массиве, состоящем из k целых элементов, вычислить:
- количество положительных элементов массива;
- сумму элементов массива, расположенных после последнего элемента, равного нулю.

9. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
- произведение отрицательных элементов массива;
- сумму положительных элементов массива, расположенных до максимального элемента.

10. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
- минимальный элемент массива;
- сумму элементов массива, расположенных между первым и последним положительными элементами.

Упражнение 2

Написать программу в соответствии с вариантом, выделив под массив динамически память. Обращаться к элементам массива необходимо используя указатель.

1. Двумерный массив, содержащий равное число строк и столбцов, называется магическим квадратом, если суммы чисел, записанных в каждой строке, каждом столбце и каждой из двух больших диагоналей, равны одному и тому же числу. Определить, является ли данный массив A из n строк и n столбцов магическим квадратом.

2. Элемент матрицы назовем седловой точкой, если он наименьший в своей строке и наибольший (одновременно) в своем столбце (или наоборот, наибольший в своей строке и наименьший в своем столбце). Для заданной целой матрицы размером 10 x 12 напечатать индексы всех ее седловых точек.

3. Дана матрица размером 6 x 6. Найти сумму наименьших элементов ее нечетных строк и наибольших элементов ее четных строк.

4. Дана целочисленная прямоугольная матрица. Определить количество строк, не содержащих ни одного нулевого элемента; максимальное из чисел, встречающихся в заданной матрице более одного раза.

5. Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

6. Характеристикой строки целочисленной матрицы назовем сумму ее положительных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик.

7. Дана целочисленная прямоугольная матрица. Определить количество столбцов, содержащих хотя бы один нулевой элемент; номер строки, в которой находится самая длинная серия одинаковых элементов.

8. Дана целочисленная квадратная матрица. Определить произведение элементов в тех строках, которые не содержат отрицательных элементов; максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

9. Для заданной матрицы размером 8 на 8 найти такие k, что k-я строка матрицы совпадает с k-м столбцом. Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

10. Характеристикой столбца целочисленной матрицы назовем сумму модулей его отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, расположить их в соответствии с ростом характеристик. Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.