CODE ACADEMY

ОБУЧЕНИЕ ПО ПРОГРАМИРАНЕ

СРЕЩА 9 Указатели и функции

int

Указател char **envp

Какво представляват аргументите на главната функция – **int main(argc, *argv[ ])**

Освен разгледаните параметри, **main** функцията има още един

char **envp - environment pointer.

Масив от стрингове, съдържащ променливите на обкръжението (environment variables):

"SSH_AGENT_PID=2292"

"USER=user" - името на потребителя

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games" - пътища т.н.

CODE ACADEMY

Указател char **envp

Напишете и компилирайте кода, за да видите променливите на обкръжението

```
#include <stdio.h>

int main(int argc, char *argv[ ], char *envp[ ])
{
      int envc = 0;
      char **p;
      for (p = envp, envc = 0; *p; p++, envc++)
         {
        printf("env \t %d \t %s \n", envc, *p);
   }

  return 0;
   }
```

CODE ACADEMY

Указател към указатели

```
int x=5;          // x е променлива
int *ptr=&x;      // ptr е променлива, която е указател към x
```

```
int ** p = &ptr;   // p е указател към указателя ptr
```

Понеже ptr е указател към int => p ще е двоен указател към int

```
int ** p;
```

```
printf("%d", *ptr);
```
→ ще изведе стойността на целочислената променлива x

```
printf("%d", **p);
```
→ прави същото

```
printf("%d", *p);
```
→ извежда адреса на клетката, където е записан указателя ptr

CODE ACADEMY

Масиви от указатели

```
int main()
{
        int *arr_ptr[3];                    // декларира масив от 3 указателя

        int a[10] ={1,2,3,4,5,6,7,8,9,10};
        arr_ptr[0]=a;
    int b[5] ={10 ,20, 30,40, 50};
        arr_ptr[1]=b;
        int c[3] ={18, 36, 54};
        arr_ptr[2]=c;

        printf("%d", arr_ptr[1][3]);

}
```

CODE ACADEMY

Създаване на многомерни масиви чрез масиви от указатели

```
int ***a_ptr;                    // декларира указател към указател към указател

int **b_ptr[3];                  // декларира масив от двойни указатели

int  *c_ptr[5];                  // декларира масив от  указатели
```

a_ptr → указател към масив с три измерения

b_ptr → указател към масив с две измерения

c_ptr → указател към масив с едно измерение

CODE ACADEMY

Динамично разпределени 2D масиви

За динамично декларираните 2D масиви може да бъде разпределена памет един от двата начина.

За двумерен масив NxM :
Разпределяме едномерна част от NxM клетки от паметта в Heap.
Разпределяме масив от масиви: разпределете 1 масив от N указатели към масиви и разпределете N масива от M мента със стойности (за всеки ред).

Може също така да се направи разпределението за всяка колона независимо и да има масив от колонни масиви.

В зависимост от метода, методите за деклариране и достъп се различават.

Динамично разпределени 2D масиви Метод 1:

int *2d_array;

2d_array = (int *) malloc(sizeof(int)*N*M);

Достъп чрез индекси

```
for(i=0; i < N; i++)
 {
    for(j=0; j < M; j++)
     {
        2d_array [i*M +j] = 0;
      }
  }
```

Достъп чрез използване на аритметика на указателите - всички N*M клетки а съседни)

```
int *ptr = 2d_array;
 for(i=0; i < N; i++) {
             for(j=0; j < M; j++) {
              *ptr = 0; ptr++;
 } }
```

Динамично разпределени 2D масиви Метод 1 – чрез Функция

```
void init2D(int *arr, int rows, int cols)
{
 int i,j;
 for(i=0; i < rows; i++) {
            for(j=0; j < cols; j++) {
             arr[i*cols +j] = 0;
} } }

 int main()
{
    int *array;
    array = malloc(sizeof(int)*N*M);

    if ( array != NULL )
 init2D(arr, N, M);

}
```

CODE ACADEMY

Динамично разпределени 2D масиви Метод 2

Разпределя се масив от N указатели към int
malloc връща адреса на този масив (указател към (int *))

```
int **2d_array;

 2d_array = (int **) malloc ( sizeof (int *) *N );
```

за всеки ред, malloc отделя пространство за неговите клетки и го добавя към масива от масиви

```
for(i=0; i < N; i++) {
  2d_array[i] = (int *) malloc(sizeof (int)*M);
}
```

CODE ACADEMY

Динамично разпределени 2D масиви Метод 2

Достъп до масивите

```
for(i=0; i < N; i++) {
  for(j=0; j < M; j++) {
     2d_array[i][j] = 0;
  }
}
```

Или

```
for(i=0; i < N; i++) {
  *ptr = 2d_array[ i ];  // поставя  pointer към адреса на клетка с индекс 0 в ред i
  for(j=0; j < M; j++) {
     *ptr = 0;
     ptr++;
  }
}
```

Двумерните масиви като параметър на функция

```
void init2D_Method2(int **arr, int rows, int cols) {
  int i, j;
  for(i=0; i < rows; i++) {
    for(j=0; j < cols; j++) {
                arr[i][j] = 0;
    }
  }
}

int main() {

 int **array;
 array = (int **) malloc ( sizeof (int *) *N );
 for(i=0; i < N; i++) {
    2d_array[i] = (int *) malloc(sizeof (int)*M);
 }

 init2D_Method2(array, N, M);
```

CODE ACADEMY

Двумерните масиви Задача за упражнение

Напишете програма, която прочита стойности в двумерен масив 3x4 с помощта на функция.

Разменят се първия и втория ред на масива.

След това стойностите на масива се извеждат

като се използва друга функция

Двумерните масиви

Функция, разменяща два реда от двумерен масив

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

void readArray(int **a,int n, int m){
 for(int k=0;k<n;k++)
    for(int l=0;l<m;l++)
        scanf("%d",&a[k][l]);
}

void printArr(int **a,int n, int m){
 for(int k=0;k<n;k++)
    {
        for(int l=0;l<m;l++)
           printf("%d\t", a[k][l]);
        printf("\n");
    }
}

int swapRow(int *x, int *y, int m){
 int row;
 for(row=0;row<n;row++){
    int tmp=x[col];
    x[row]=y[row];
    y[row]=tmp;
  }
}
```

```
int main() {
int i, N=3,M=4;
 int **array;

 array = (int **) malloc ( sizeof (int *) *N );

 for(i=0; i < N; i++) {
    array[i] = (int *) malloc(sizeof (int)*M);
  }

  readArray(array, N, M);
  printArr(array,N,M);

  int maximum = maxElem(array, N, M);

  for(i=0; i < N; i++) {
     free( array[i]);
     array[i]=NULL;
  }

  free(array);
  array=NULL;

}
```

Двумерните масиви

Функция, връщаща максималните елементи за всеки един ред от двумерен масив

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

////////////////////////////////////////
void readArray(int **a,int n, int
m){
 for(int k=0;k<n;k++)
    for(int l=0;l<m;l++)
       scanf("%d",&a[k][l]);
}
////////////////////////////////////////
void printArr(int **a,int n, int m){
 for(int k=0;k<n;k++)
    {
       for(int l=0;l<m;l++)
          printf("%d\t", a[k][l]);
       printf("\n");
    }
}
int * maxElem(int **a, int n, int
m){

       int * m=(int *) malloc(sizeof
(int)*M);;

       int row, col;
}
```

```
int main() {
int i, N=3,M=4;
 int **array;

 array = (int **) malloc ( sizeof (int *) *N
);

 for(i=0; i < N; i++) {
     array[i] = (int *) malloc(sizeof
(int)*M);
  }

  readArray(array, N, M);
  printArr(array,N,M);
}
```

CODE ACADEMY

Selection Sort

Algorithm for Selection Sort:

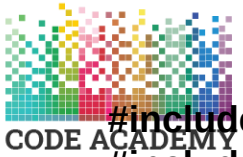
Step 1 − Set min to the first location

Step 2 − Search the minimum element in the array

Step 3 – swap the first location with the minimum value in the array

Step 4 – assign the second element as min.

Step 5 − Repeat the process until we get a sorted array.



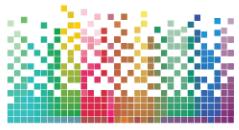
Selection Sort

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

void readArray(int *a, int n){
 for(int i=0;i<n;i++)
    scanf("%d",a+i);
}

void printArray(int *a, int n){
 for(int i=0;i<n;i++)
    printf("%d\t",*(a+i));
}

void selectionSort(int *a, int n){
    int m;
  for(int j=0; j< n-1; j++){
    m=j;
    for(int k=j+1;k<n;k++){
        if(a[m]>a[k]){
            m=k;
        }
    }
    int tmp=a[m];
    a[m]=a[j];
    a[j]=tmp;

    }
  }
```

Selection Sort

```
int main() {

 int N=10;

 int *array;
 array = (int *) malloc(sizeof (int)*N);

 readArray(array, N);
 printArray(array,N);

 selectionSort(array,N);
  printArray(array,N);


}
```

Insertion Sort



Сортирането чрез вмъкване е прост алгоритъм за сортиране за малък брой елементи.
В сортиране с вмъкване сравнявате ключовия елемент с предишните елементи.
Ако предишните елементи са по-големи от ключовия елемент,
тогава премествате предишния елемент на следващата позиция.

Започва се от индекс 1 докато се достигне размера на входния масив.
[8 3 5 1 4 2]
key = 3 //starting from 1st index.
Тук key ще бъде сравнен с предишните елементи. В този случай „ключът“ се сравнява с 8. тъй като 8 > 3,
преместваме елемент 8 до следващата позиция и вмъкваме key в предишната позиция.
Result: [3 8 5 1 4 2]
Key = 5 --> *индекс 1*
8 > 5 --> преместваме 8 на 2-ра позиция и вмъкваме 5 на 1-ва позиция.
Result: [3 5 8 1 4 2]
Key = 1 --> трети индекс
8 > 1 => [3 5 1 8 4 2]
5 > 1 => [3 1 5 8 4 2]
3 > 1 => [1 3 5 8 4 2]
Result: [1 3 5 8 4 2]
key = 4 --> четвърти индекс
8 > 4 => [1 3 5 4 8 2]
5 > 4 => [1 3 4 5 8 2]
3 > 4 ≠> stop
Result: [1 3 4 5 8 2]

CODE ACADEMY

Insertion Sort

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

CODE ACADEMY

Quick Sort

Quicksort е много ефективен метод за сортиране. Ние последователно разделяме списъка на по-малки списъци и прилагаме същият алгоритъм към подсписъка.

Алгоритъм:
Разглеждаме един елемент в даден момент (пивот) и го поставяме в правилната му позиция.Всички елементи отляво на опората трябва да са по-малки от опората и всички елементи вдясно да са по-големи.

Това разделя масива на две части - ляв дял и десен дял. Същият метод се прилага за всеки дял. Процесът продължава, докато не може да се направи повече дял.
Всеки елемент може да бъде избран като опорна точка.

Рекурсивната функция е подобна на Mergesort, но не се разделя на две равни части, а се разделя на база на основен елемент.

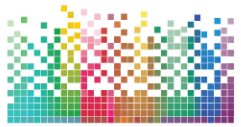
CODE ACADEMY

Quick Sort

```
void Quicksort( int a[], int lb,int ub)
{
        int j;
        if(lb<ub)
        {
                j=Partition(a,lb,ub); //partition the array
                Quicksort(a,lb,j-1); //sort first partition
                Quicksort(a,j+1,ub); //sort second partition
        }
}
```

ub=горна граница
lb=долна граница

https://www.studytonight.com/data-structures/quick-sort



Quick Sort – с рекурсия

```c
#include <stdio.h>

void swap(int *n1, int *n2) {
  int m = *n1;
  *n1 = *n2;
  *n2 = m;
}

int partition(int arr[], int x, int y) {

  int pivot = arr[y];

  int i = (x - 1);

  for (int j = x; j < y; j++) {
    if (arr[j] <= pivot) {

      i++;

      swap(&arr[i], &arr[j]);
    }
  }

  swap(&arr[i + 1], &arr[y]);

  return (i + 1);
}
```

```c
void quickSort(int arr[], int x, int y) {

  if (x < y) {

    int pi = partition(arr, x, y);

    quickSort(arr, x, pi - 1);

    quickSort(arr, pi + 1, y);
  }
}

void printArray(int arr[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", arr[i]);
  }
  printf("\n");
}

int main() {
  int values[] = {99, 88, 64, 25, 40, 20, 42};

  int n = sizeof(values) / sizeof(values[0]);

  printf("Given Unsorted Array Values \n");
  printArray(values, n);

  quickSort(values, 0, n - 1);

  printf("Sorting Given Array In The Ascending Order: \n");
  printArray(values, n);
}
```

CODE ACADEMY

Binary Search

```
int binarySearch(int *array, int x, int low, int high) {
  // Repeat until the pointers low and high meet each other

  while (low <= high) {

    int mid = low + (high - low) / 2;

    if (array[mid] == x)
        return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}
```

Binary Search

```
int main() {
  int array[ ] = {3, 4, 5, 6, 17, 38, 90};

  int n = sizeof(array) / sizeof(array[0]);

  int x = 4;

  int result = binarySearch(array, x, 0, n - 1);

  if (result == -1)
        printf("Not found");
  else
        printf("Element is found at index %d", result);

  return 0;
}
```

CODE ACADEMY

Вградени функции

Предимство на функциите е, че техния код се съхранява само на едно място в паметта, независимо от това колко пъти се извикват. Така се намалява памет.

Икономията на памет е за сметка на бързодействието и обратно.

Всяко обръщане към функция е свързано с изпълнение на служебни операции, като запомняне адреса на връщане, предаване на параметри и др.

Има програми, в които е важно да има максимално бързодействие. Такива са - програми за управление на различни устройства и програми за управление на технологични процеси. Такива програми работят в реално време и работата им се съобразява с различен контекст, като ограничение по време, временни ограничения, предизвикани от обекти и др.

За да повиши бързодействието, езика С поддържа така наречените **вградени функции** или **Inline functions**

CODE ACADEMY

Inline functions - Вградени функции

Кодът на тези функции се копира на всяко място в паметта, където има обръщение към тях. По този начин се увеличава бързодействието им, защото броят на служебните операции е малък.

Дефинират се и се използват както всички други функции.

Пред дефиницията на функцията се записва ключовата дума inline

```
inline void bell();
```

Автоматични променливи

Те биват параметри на функции и стандартни променливи.

Автоматичните променливи имат следните основни характеристики:

- Те са **локални за функцията** и са видими само вътре
- Те се **създават при всяко извикване на функцията**

В различните функции могат да се използват автоматични променливи с еднакви имена, защото имат локален характер

Могат да се инициализират при тяхното дефиниране и тогава те се инициализират отново, когато функцията бъде извикана, защото са локални.

Пример:

```
void f(int x, int y){
   float p=3.56;…}
```

Паметта за автоматичните променливи се заделя в стека!

Използването на автоматични променливи икономисва памет!

Параметрите на функциите предавани по стойност са автоматични променливи.

Автоматични променливи

Всички локални променливи, които са дефинирани във функцията, са известни като auto (автоматични) променливи, освен ако не са посочени, т.е. по подразбиране локалната променлива е автоматична променлива. Няма нужда да поставяте ключовата дума auto (тя е по избор), докато декларирате локална променлива.

Всеки път, когато функцията, в която е декларирана променлива, се извиква и се унищожава, когато изпълнението на програмата напусне функцията се създава нова автоматичната променлива.

```
#include <stdio.h>
int main(){
int a;
auto int b;
a=10; b=20;
printf("a=%d, b=%d\n",a,b);
return 0;
}
```

CODE ACADEMY

Указател към функция

Тип на върната стойност (* име) (параметри на функцията);

Примери

int (* fptr) ( );

void (* funcptr) (int *x, double y);

(* fptr) и (* funcptr) са указател към функция

int * f1ptr ( ) → функция, връщаща указател

CODE ACADEMY

Указател към функция

1. Указва изпълним код
2. Подобно на указателите към данни,
   можем да разгледаме указател към машинен код

```
int myfunc(double, int); /* prototype function */
void fnPointerDemo() {
  int (*ptrfun) (double, int);
  ptrfun = myfunc;
  int nValue = (*ptrfun) (1.0, 1);
}

int mystery(int a, int b, int (*fn)(int,int)) {
 return ((*fn)(a,b));
}
```

CODE ACADEMY

Пример

```
double myfunc(double a, int b)
{
   return (a * b);
}

double  fnPointerDemo()
{
  double (*ptrfun) (double, int);
  ptrfun = myfunc;
  double dValue = (*ptrfun) (4.5, 1);

  return dValue;
}

int main()
{
   printf("result = %f", fnPointerDemo());

}
```

CODE ACADEMY

Указател към функция

int *ap[10]; декларира масив от 10 указателя.

float *fp(float);
декларира функция, която получава float като аргумент
и връща float пойнтер.

void (*pf)(int);
декларане на указател към функция,която
получава като аргумент int променлива
и тази функция не връща нищо, защото типът, който връща функцията, е void.

int *(*x[10])(void);
масив от 10 указателя към функции, които не получават параметри,
а типът, който връщат пойнтери към функция е указател от тип int.

typedef (* fpointer)(argument list);

CODE ACADEMY

Указател към функция-Пример

```
#include<stdio.h>

void tabul( float (*f)(float), float x[], float y[] , int count, float step)
{
    for(int i=0;i<count;i++){
        x[i]=i*step;
        y[i]=(*f)(x[i]);
    }
}

float f1(float x){
    return x*x+2;
}

float f2(float x){
    return x*x-1;
}

int main(){
    float x[5], y[5];

    tabul(f1, x, y, 5, 2);

    for(int i=0; i<5;i++){
        printf("x[%d]=%f\t y[%d]=%f\n", i, x[i],i,y[i]);
    }
 printf("\n\n");
     tabul(f2, x, y, 5, 3);

    for(int i=0; i<5;i++){
        printf("x[%d]=%f\t y[%d]=%f\n", i, x[i],i,y[i]);
    }}
```

CODE ACADEMY

Указател към функция

Задача 1. Направете две функции и извикайте желаната функция с указател към функция, съобразно знака, подаден от командния ред:

Пример: a.exe 20 + 3

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

int fnPlus(int nA, int nB) { return(nA + nB); }
int fnMinus(int nA, int nB) { return(nA - nB); }

int ( * pfCalc ) ( int, int ) = NULL;
```

Задача 2. Добавете към горния код функции за умножение и деление.

CODE ACADEMY

Указател към функция

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

int fnPlus(int nA, int nB) { return(nA + nB); }
int fnMinus(int nA, int nB) { return(nA - nB); }

int ( * pfCalc ) ( int, int ) = NULL;

int main(){
char op;
scanf("%c",&op);

int a=4;
int b=9;

if(op=='+')
    pfCalc =fnPlus(a,b);
    printf("%d", ( * pfCalc )   );
}
```

Да се довърши задачата за домашна работа

CODE ACADEMY

Указател към функция

Задача 3. Направете масив от указатели към функции по следния начин:

```
void add(){…}

void fun2(){…}

void fun3(){...}

void (*func_ptr[3])() = {fun1, fun2, fun3};
```

Направете меню, с което да питате потребителя коя от горните функции иска да използва – събиране, изваждане, умножение и деление.

След това попитайте за числата, които да участват в тази операция, и извикайте функцията, която да извърши желаната операция.

CODE ACADEMY

Указател към функция

Задача 4. Напишете функция **void sort_arr(void *A, int n, int dir, ftype fp)**,

където A е обикновен масив от цели числа,

n - размера на масива,

dir - посоката, в която да бъде сортиран масива,

ftype - указател към функция, която приема масив, неговия размер и в каква посока да бъде сортиран входния масив.

ftype трябва да сортира масива във възходящ или низходящ ред в зависимост от третия параметър на функцията.

Принтирайте изходния и резултатни масиви.

CODE ACADEMY

Решението на задача 7 от домашното

https://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/

CODE ACADEMY

Задачата от 20 май за обединяване на масиви

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

void swap(int *p, int *q){
   int tmp=*p;
   *p=*q;
   *q=tmp;
}


void bubbleSort(int x[], int n)
{
     int br=0;
     while(br<n)
     {
        for(int i=0; i<n-1; i++){
             if(x[i+1]>x[i])
                 swap(&x[i],&x[i+1]);
        }
      br++;
     }
}

void printArr(int w[], int n){
   for(int ind=0;ind<n; ind++)
       printf("%d\t", *(w+ind));
   printf("\n");
}
```

```c
int main()
{
 int a[3]={1,5,8};
 int b[5]={10,25,18,2,34};
 bubbleSort(a,3);
 printArr(a,3);
 bubbleSort(b,5);
 printArr(b,5);

 int *c=malloc(8*sizeof(int))

 int nA=3;
 int nB=5;


 int i=0,j=0,k=0;

 while(i<nA && j<nB){
      if(a[i]>b[j])
      {
           c[k]=a[i];
           i++;
           k++;
      }
      else
      {
          c[k]=b[j];
           j++;
           k++;
      }
 }
```

```c
while(i<nA)
{
     c[k]=a[i];
     i++;
     k++;
}

while(j<nB)
{
     c[k]=b[j];
     j++;
     k++;
}

printArr(c,8);


}
```