# Обработка на масиви. Работа с масиви и указатели. Обмен на данни (базови типове, масиви) между функции. Разработване на програми с масиви.

# Динамични масиви

В езика С няма предвидена опция за работа с масив от динамично променящи се елементи, но използвайки указатели и някои опции на езика може да реализираме такъв похват и да работим с променливи данни.

Динамичният масив представлява масив, чиято дължина не е известна преди началото на изпълнение на програмата и може големината да бъде променяна в хода на изпълнение на програмата. Динамичните масиви се използват в много случаи, когато се налага записването на предварително неизвестен брой елементи, както и когато е нужно намаляване на заеманата памет с цел оптимизация на заеманите системни ресурси.

За заделянето/презаделянето/освобождаването на динамични масиви се използват функции от стандартната библиотека stdlib.h

Функциите имат следния вид:

void\* malloc(size\_t bytes);

void\* calloc(size\_t elements, size\_t bytes\_per\_element);

void\* realloc(void\* prev\_address, size\_t new\_size);

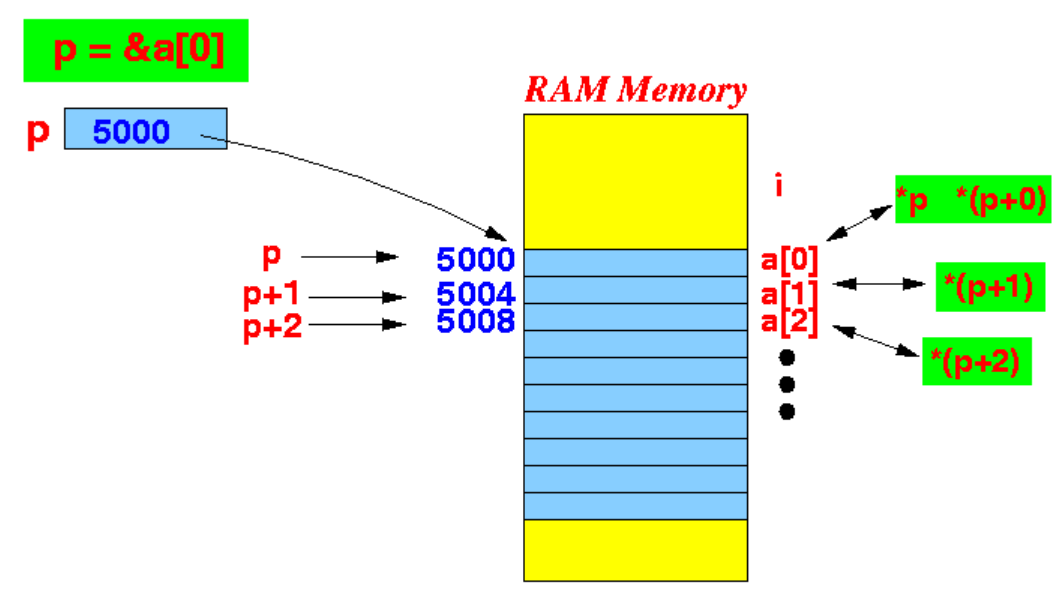
void free(void\* array);

В дефиницията на функциите са използвани следните типове:

**void\*** - без типов указател това означава, че няма информация за типа на данните, които са разположени на даден адрес в паметта;

**size\_t** – предефиниран тип от unsigned long – неотрицателна стойност за задаване на размер.

На следващото изображение може да видим логиката за работа на масив в паметта. В случая, когато заделим блок памет с определен размер, може този блок да го разбием логически на елементи чрез свойството на указателите да се движат през определен тип данни като по този начин симулираме работата на масив.



malloc()

Функцията malloc() изисква следните параметри:

***void \* malloc();***

Функцията заделя определения брой байтове и връща указател към адреса на началото на тези заделени данни. „Заделени данни“ означава, че този ресурс памет е специално заделен за използване от програмата и няма да бъде достъпван от външни програми, както и няма променливи, които да използват данните на заделеното пространство. Malloc НЕ изтрива данните, а само ги заделя. Програмистът е длъжен да нулира данните предварително за дейности като броене и др.

calloc()

Функцията calloc() изисква следните параметри:

***void\* calloc(,);***

Фукнцията заделя и нулира заделената памет. Тази функция е предназначена за заделяне на памет за масиви, тъй като приема два параметъра – броят на елементите в масива и големината в байтове на всеки един елемент. Резултатът, който връща е указател към първия елемент на масива.

realloc()

Функцията realloc() изисква следните параметри:

***void\* realloc(,);***

Функцията има за цел да промени размера на текущо заделената памет за динамичния масив, като приема два параметъра: старият адрес, където са разположени данните и големината на масива след промяна на размера. Новият размер може да бъде по-голям или по-малък от стария.

Realloc копира (ако е нужно) данните от стария масив в новия масив, до там докъдето новата големина. Функцията връща като резултат новото местоположение на данните.

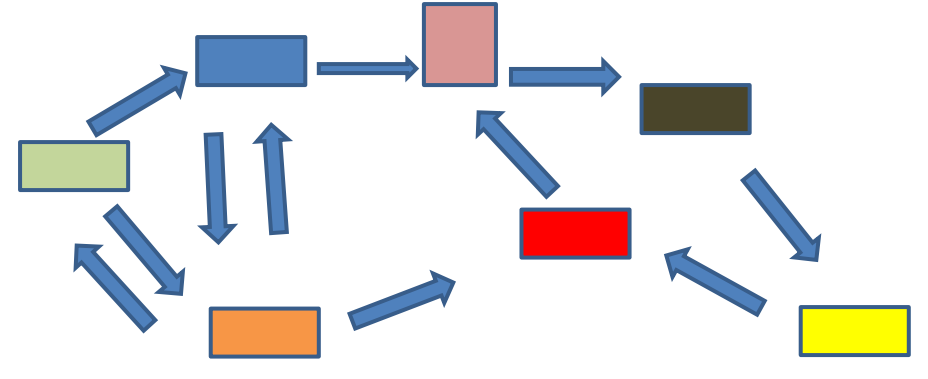
Всички функции по-горе връщат като резултат NULL, ако не успеят да заделят памет с изисквания размер. Важно е да се проверява дали резултатът им е NULL, понеже може да се появи проблем при записването на данни на нерегламентирани места и да се получи конфликт на данните.

free()

Тази функция освобождава заетата от по-горните функции памет, като приема само един параметър – адресът на началото на динамичния масива. Добър стил на писане е, ако в края на програмата програмистът освободи заделената памет. Добре е да се внимава, кога се освобождава памет, понеже двойно освободената памет може да предизвика конфликти, както и не навременното освобождаване да предизвика недостиг на памет. В други езици от по-висок клас има вградени подсистеми, които се грижат за автоматичното освобождаване на ненужно заета памет. Тъй като програмният един С е предвиден да е език от средно ниво, то тези функции са поверени на програмиста.

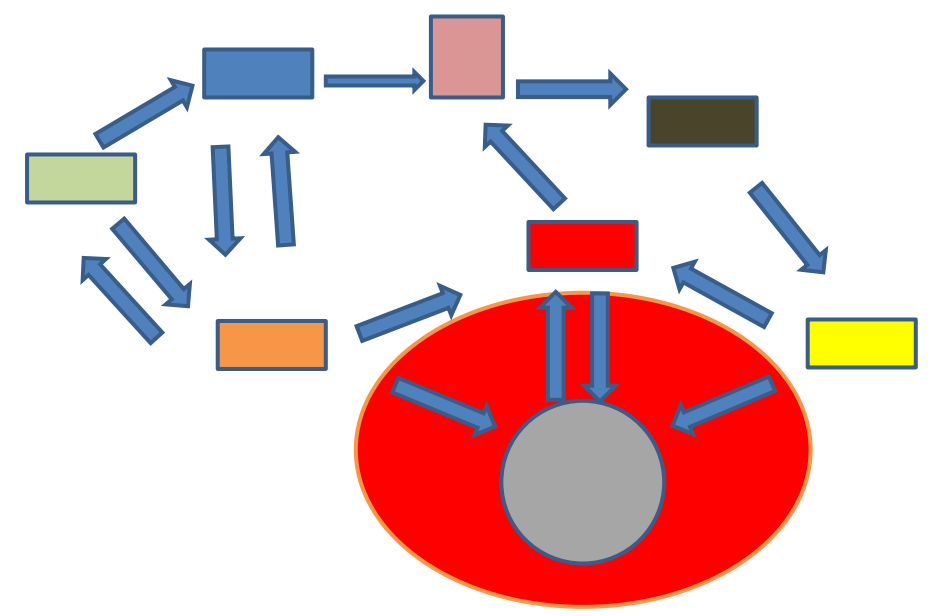
Ако все още не сте разбрали какво е динамичната памет, и къде ни е от полза, то конкретният пример ще внесе яснота.

Представете си, че имаме релсова пътна инфраструктура, която трябва да бъде изградена. От фирма „Х“ ни дават нужният маршрут, който ние да симулираме и стигаме до следният вариант:



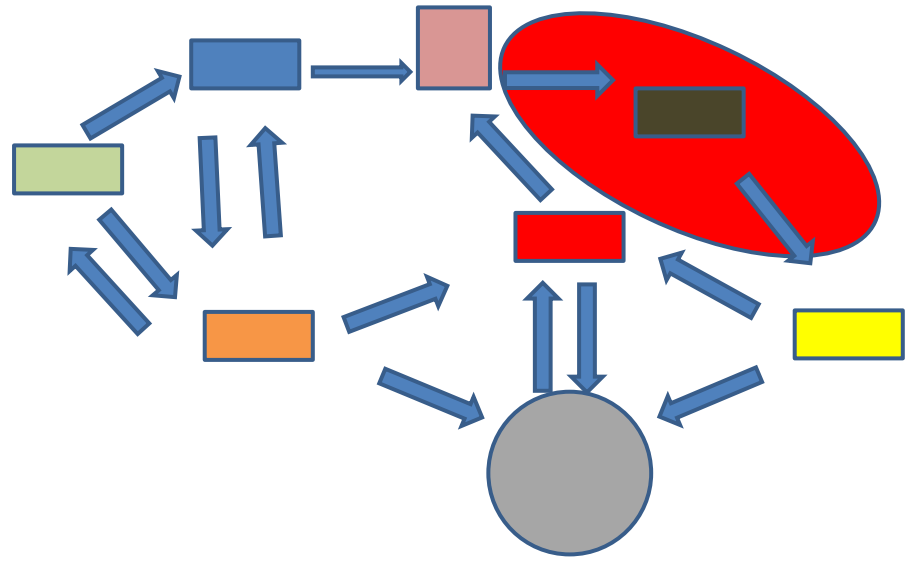
Както се вижда на графиката по-горе до всяка гара (правоъгълник) може да се стигне по определен път. Гарите са 7 на брой, а изградените релси 11. Правейки си статичен масив и работейки с тези числа, имате фиксирана система, която изпълнява своите функции, но както в реалния живот ако трябва да разширим някой от критериите (релси или гара) няма да можем (до момента масивът ни, който държи информация за гарите и за релсите е с точно определена дължина – 7 за гари и 11 за релси ).

Идва ни заявка за нова гара. Също така за 4 релси които я свързват с другите(в червеното поле).



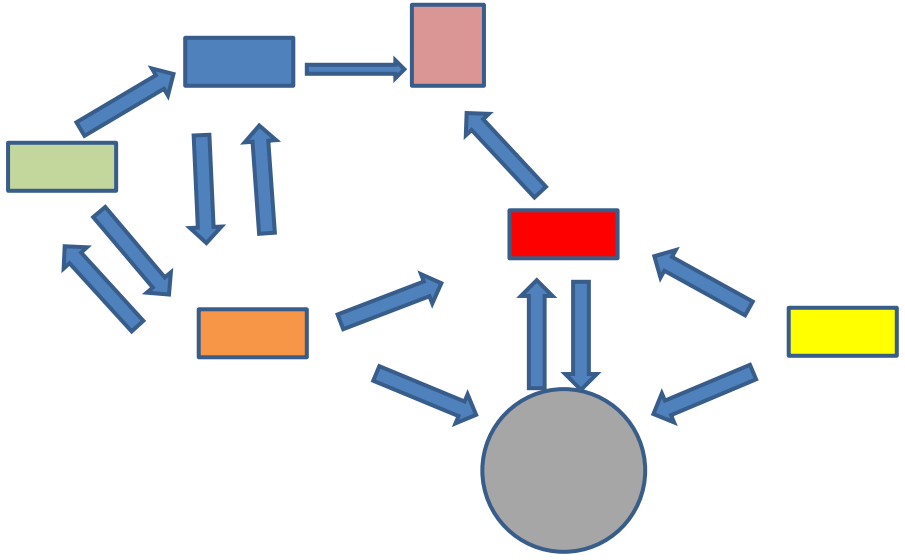
Ако не съществуваха динамичните данни единственият вариант за промяна и въвеждане на новите данни е пренаписване на цялата програма от програмист. Тук идва и силата на разглежданата техника. Използвайки функция за допълнително заделяне на памет можем да вземем място за новата гара и четирите релси във вече съществуващите масиви (стига да имаме достатъчно памет). Така само чрез избирането на опция „добави“, администраторът на приложението може да конфигурира цялата си инфраструктура без намесата на трети лица.

Какво ще се случи ако гара или релси са вече неизползваеми? Решаваме да премахнем една гара с пътищата към нея.

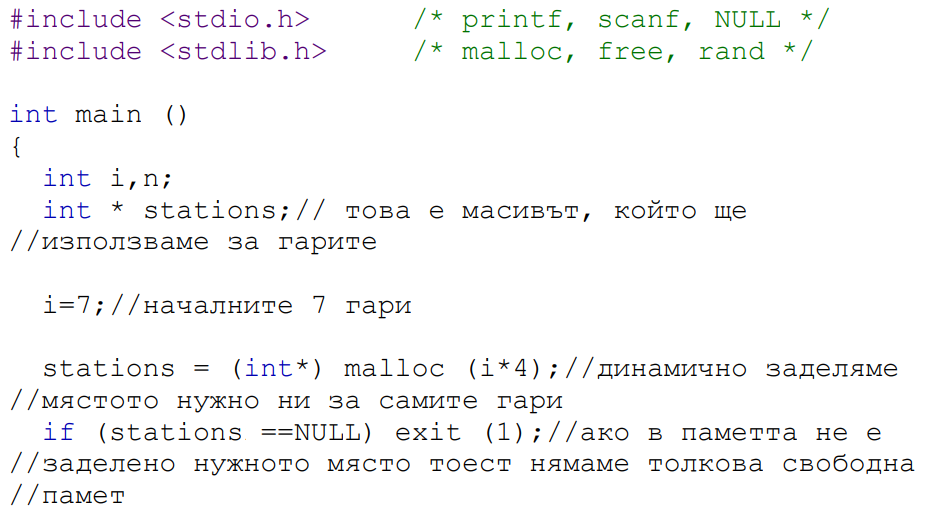


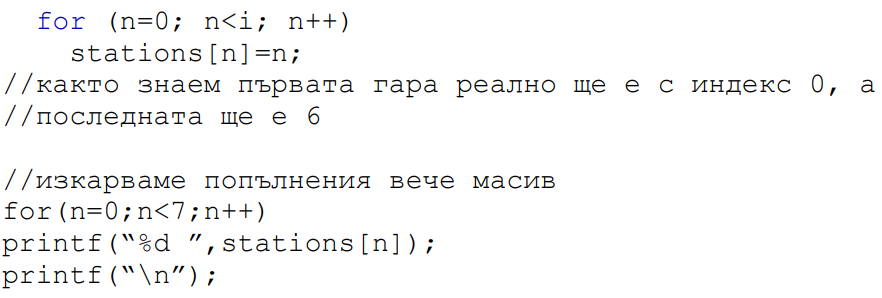
В масива от данни, който имаме можем просто да кажем че този елемент го няма вече (да го отбележим с -1), но така остава заделена големината на масива и реално имаме излишък на заето пространство. Тук идва и метода за освобождаване на памет - щом може да бъде заделена динамично, то логично е и да може да бъде освободена по същият начин.

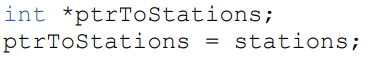
Крайният резултат ще бъде следният:

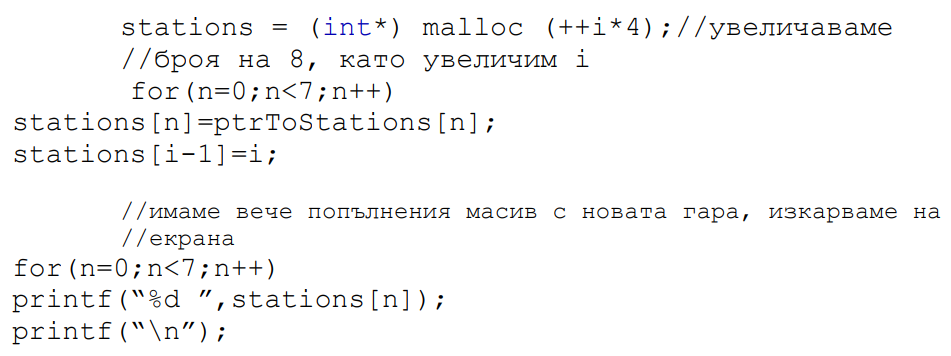


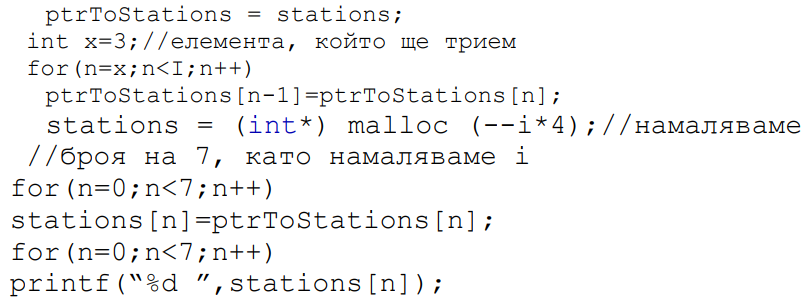
Примери за решение на задачата с гореописаните функции:

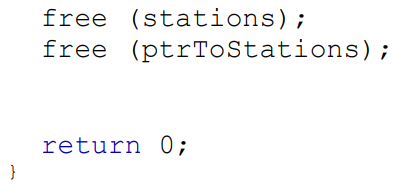




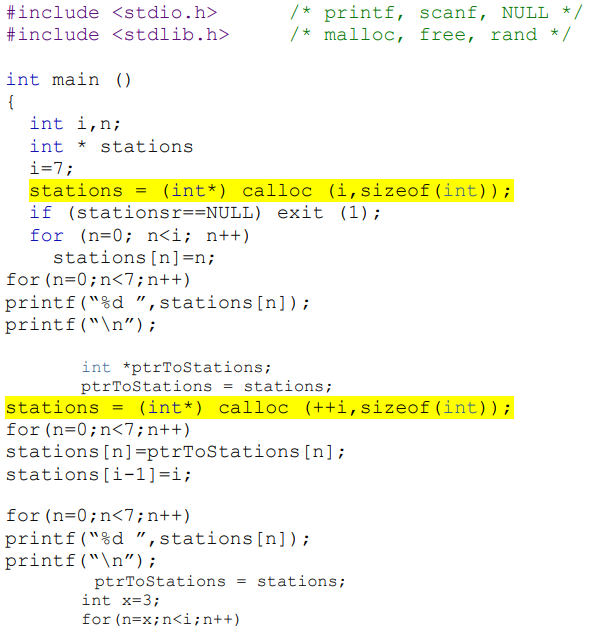


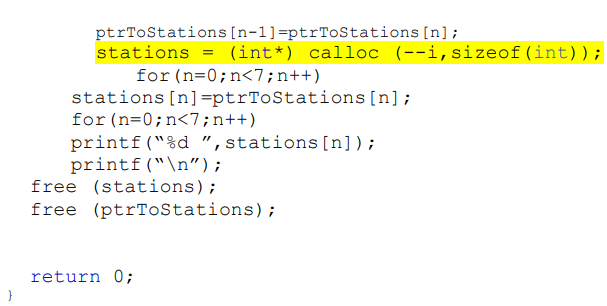




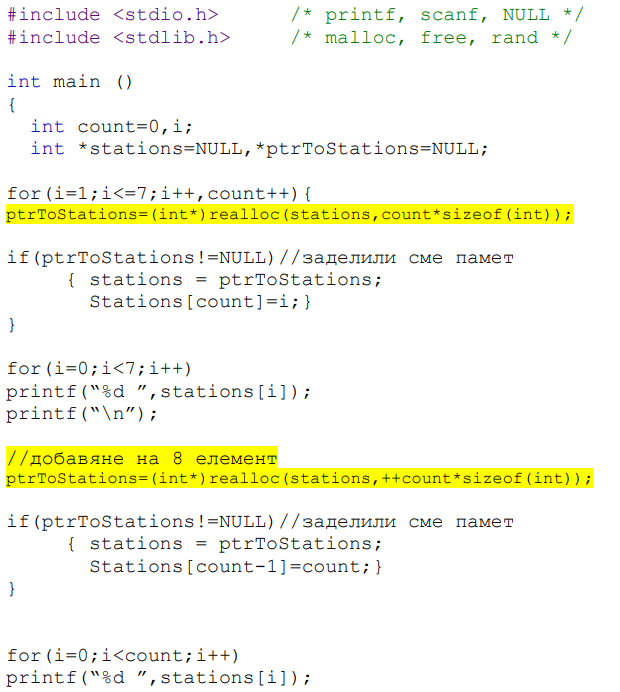


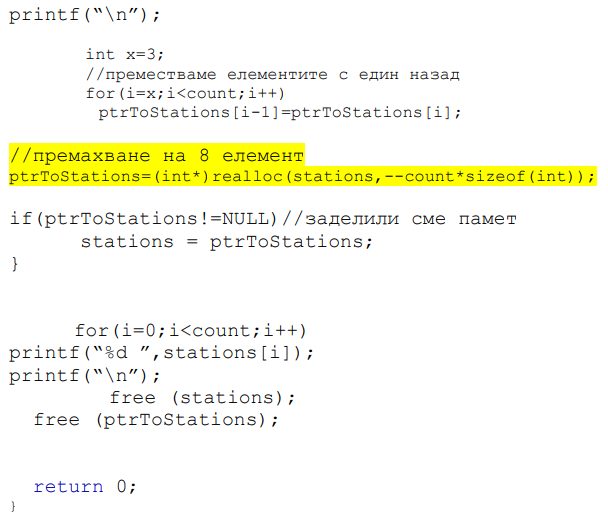
Реализация с calloc():





Реализация с realloc():





Код работещ с динамични масиви и реални числа:

**int main(int argc, char \*argv[])**

**{**

**int i;**

**double\* p;**

**p = calloc(10, sizeof(double) ); // масив с реални 10 елемента**

**for ( i = 0; i < 10; i++ )**

**\*(p + i) = i; // стойност i слагаме в масива с индекс i**

**for ( i = 0; i < 10; i++ )**

**printf("\*(p + %d) = %lf\n", i, \*(p+i) );**

**free(p); // освбождаване на памет**

**putchar('\n');**

**p = calloc(4, sizeof(double) ); // правим НОВ масив с 4 елемента**

**// \*\*\*\*\* Забележете, размера на масива е променен !!! \*\*\*\***

**for ( i = 0; i < 4; i++ )**

**\*(p + i) = i\*i; // слагаме i\*i на индекс i**

**for ( i = 0; i < 4; i++ )**

**printf("\*(p + %d) = %lf\n", i, \*(p+i) );**

**free(p); // освобождаваме памет**

**}**

Създаване на двумерен масив чрез динамична памет

Използване на едномерен масив с аритметика на указатели:

#include <stdio.h>

#include <stdlib.h>

**int** main(**void**)

{

**int** r = 3, c = 4;

**int**\* ptr = **malloc**((r \* c) \* **sizeof**(**int**));

    /\* Putting 1 to 12 in the 1D array in a sequence \*/

**for** (**int** i = 0; i < r \* c; i++)

        ptr[i] = i + 1;

    /\* Accessing the array values as if it was a 2D array \*/

**for** (**int** i = 0; i < r; i++) {

**for** (**int** j = 0; j < c; j++)

**printf**("%d ", ptr[i \* c + j]);

**printf**("\n");

    }

**free**(ptr);

**return** 0;

}

Създаване на масив от указатели, които да разширим с новозаделени елементи

#include <stdio.h>

#include <stdlib.h>

**int** main()

{

**int** r = 3, c = 4, i, j, count;

**int**\* arr[r];

**for** (i = 0; i < r; i++)

        arr[i] = (**int**\*)**malloc**(c \* **sizeof**(**int**));

    // Note that arr[i][j] is same as \*(\*(arr+i)+j)

    count = 0;

**for** (i = 0; i < r; i++)

**for** (j = 0; j < c; j++)

            arr[i][j] = ++count; // Or \*(\*(arr+i)+j) = ++count

**for** (i = 0; i < r; i++)

**for** (j = 0; j < c; j++)

**printf**("%d ", arr[i][j]);

    /\* Code for further processing and free the

      dynamically allocated memory \*/

**for** (**int** i = 0; i < r; i++)

**free**(arr[i]);

**return** 0;

}

Използване на указател към указател като направим динамичен масив с указатели и после заделим памет за всеки от елементите като ползваме двоен указател:

#include <stdio.h>

#include <stdlib.h>

**int** main()

{

**int** r = 3, c = 4, i, j, count;

**int**\*\* arr = (**int**\*\*)**malloc**(r \* **sizeof**(**int**\*));

**for** (i = 0; i < r; i++)

        arr[i] = (**int**\*)**malloc**(c \* **sizeof**(**int**));

    // Note that arr[i][j] is same as \*(\*(arr+i)+j)

    count = 0;

**for** (i = 0; i < r; i++)

**for** (j = 0; j < c; j++)

            arr[i][j] = ++count; // OR \*(\*(arr+i)+j) = ++count

**for** (i = 0; i < r; i++)

**for** (j = 0; j < c; j++)

**printf**("%d ", arr[i][j]);

    /\* Code for further processing and free the

       dynamically allocated memory \*/

**for** (**int** i = 0; i < r; i++)

**free**(arr[i]);

**free**(arr);

**return** 0;

}

Използване на двоен указател и едно викане на malloc()

#include<stdio.h>

#include<stdlib.h>

**int** main()

{

**int** r=3, c=4, len=0;

**int** \*ptr, \*\*arr;

**int** count = 0,i,j;

    len = **sizeof**(**int** \*) \* r + **sizeof**(**int**) \* c \* r;

    arr = (**int** \*\*)**malloc**(len);

    // ptr is now pointing to the first element in of 2D array

    ptr = (**int** \*)(arr + r);

    // for loop to point rows pointer to appropriate location in 2D array

**for**(i = 0; i < r; i++)

        arr[i] = (ptr + c \* i);

**for** (i = 0; i < r; i++)

**for** (j = 0; j < c; j++)

            arr[i][j] = ++count; // OR \*(\*(arr+i)+j) = ++count

**for** (i = 0; i < r; i++)

**for** (j = 0; j < c; j++)

**printf**("%d ", arr[i][j]);

**return** 0;

}

Използване на указател към динамично променлив масив:

Размерността на масива са свързани с типа на роменливите. Указателя трябва да се дереференсира чрез (\*arr)[i][j]

#include <stdio.h>

#include <stdlib.h>

**int** main()

{

**int** row = 3, col = 4, i, j, count;

**int** (\*arr)[row][col] = **malloc**(**sizeof** \*arr);

    count = 0;

**for** (i = 0; i < row; i++)

**for** (j = 0; j < col; j++)

            (\*arr)[i][j] = ++count;

**for** (i = 0; i < row; i++)

**for** (j = 0; j < col; j++)

**printf**("%d ", (\*arr)[i][j]);

**free**(arr);

**return** 0;

}

Използване на указател към първия ред на динамичния масив – подобно на горното решение, но може да ползваме синтаксиса за нормален двумерен масив.

#include <stdio.h>

#include <stdlib.h>

**int** main()

{

**int** row = 3, col = 4, i, j, count;

**int** (\*arr)[col] = **calloc**(row, **sizeof** \*arr);

    count = 0;

**for** (i = 0; i < row; i++)

**for** (j = 0; j < col; j++)

            arr[i][j] = ++count;

**for** (i = 0; i < row; i++)

**for** (j = 0; j < col; j++)

**printf**("%d ", arr[i][j]);

**free**(arr);

**return** 0;

}