УРОК 04

1

Добро утро на всички! Днес се гмуркаме във вълнуващия свят на **масивите** в езика за програмиране C. **Масивите са фундаментална структура от данни**, която ни позволява да съхраняваме и манипулираме колекции от елементи от един и същи тип. Те са невероятно гъвкави и могат да се използват за ефективно решаване на широк спектър от програмни проблеми.

И така, какво точно е масив? **С прости думи, масивът е непрекъснат блок от памет, който може да съдържа фиксиран брой елементи**. Всеки елемент в масива е достъпен чрез неговия индекс, който представлява неговата позиция в масива. Индексът на първия елемент винаги е 0, а индексът на последния елемент е размерът на масива минус единица.

Нека започнем с деклариране на масив в C:

```
int numbers[5];  // Declaration of an integer array with 5 elements
```

В този пример сме декларирали масив, наречен, numbersкойто може да съхранява пет цели числа. По подразбиране елементите на масива са неинициализирани, което означава, че съдържат ненужни стойности. Ако искаме да инициализираме елементите, можем да го направим по време на декларацията:

```
int numbers[5] = {10, 20, 30, 40, 50};  // Initializing array elements
```

Сега, след като имаме нашия масив, как да получим достъп до неговите елементи? Както споменах по-рано, достъпът до всеки елемент се осъществява чрез неговия индекс. Например, за достъп до първия елемент от numbersмасива, ще използваме следния синтаксис:

```
int firstNumber = numbers[0];  // Accessing the first element
```

Важно е да се отбележи, че индексите на масива започват от 0 и достигат до size - 1. Достъпът до елемент извън валидния диапазон на индекса води до недефинирано поведение, така че трябва да внимаваме да не превишаваме границите на масива.

Масивите могат да се използват и в цикли за извършване на повтарящи се действия върху множество елементи. Да кажем, че искаме да отпечатаме всички елементи от нашия numbersмасив. Можем да използваме цикъл, като например forцикъла, за итериране на масива:

```
for (int i = 0; i < 5; i++) {
printf("%d ", numbers[i]);
}
```

Този цикъл започва с i = 0и продължава до i < 5, като се увеличава іс 1 във всяка итерация. Вътре в цикъла използваме numbers[i]за достъп до всеки елемент от масива и го отпечатваме с помощта на printfфункцията. По този начин можем да отпечатаме всички елементи на numbersмасива.

Можем също да модифицираме елементите на масив, като им присвоим нови стойности. Да кажем, че искаме да променим стойността на втория елемент в numbersмасива на 100. Можем да го направим по следния начин:

```
numbers[1] = 100;  // Modifying the second element
```

Масивите са невероятно мощни, защото ни позволяват да извършваме операции върху множество елементи едновременно. Например, да приемем, че искаме да намерим сумата от всички елементи в numbersмасива. Можем да използваме цикъл, за да итерираме масива, като добавяме всеки елемент към текущия сбор:

```
int sum = 0;
for (int i = 0; i < 5; i++) {
sum += numbers[i];
}
printf("Sum: %d\n", sum);
```

		В този кодов фрагмент инициализираме променлива sumна 0. Вътре в цикъла добавяме всеки елемент от масива към sumизползване на +=оператора. Накрая отпечатваме сумата с printf.
		Това обхваща основите на масивите в C. Научихме как да декларираме масиви, да осъществяваме достъп до техните елементи с помощта на индекси, да модифицираме елементите и да извършваме операции върху тях с помощта на цикли. Масивите са фундаментална концепция в програмирането и овладяването им ще отвори свят от възможности за ефективно решаване на сложни проблеми.
		Насърчавам всички ви да експериментирате с масиви и да изследвате допълнително възможностите им. В следващия ни урок ще се потопим по-дълбоко в разширените операции с масиви, като например многомерни масиви и манипулиране на низове. Благодаря ви за вниманието и ще се видим следващия път!
	2	Добър ден на всички! Днес ще изследваме очарователната връзка между указатели и масиви в езика за програмиране C. Разбирането на тази връзка е от решаващо значение за изграждането на мощни и ефективни програми. И така, нека се потопим!
		Първо, нека обобщим какво знаем за масивите. Масивът е непрекъснат блок от памет, който съхранява фиксиран брой елементи от един и същи тип. Всеки елемент в масива може да бъде достъпен чрез неговия индекс. Вече видяхме как се декларират и работят с масиви в предишния ни урок.
		Сега нека представим указатели. Указателят е променлива, която съхранява адреса на паметта на друга променлива. С други думи, той "посочва" мястото, където дадена стойност се съхранява в паметта. Указателите се обозначават със * символа в C.
		И така, как са свързани масивите и указателите? Е, масивите и указателите имат тясна връзка. Всъщност масивите и указателите често могат да се използват взаимозаменяемо в много контексти.
		Когато декларираме масив, името на самия масив представлява указател към първия елемент на масива. Нека да разгледаме един пример:
		int numbers[5] = {10, 20, 30, 40, 50};
		int *ptr = numbers; // Assigning the address of the first element to a pointer

		В този кодов фрагмент ние декларираме извикан масив numbersи го инициализираме с някои стойности. След това декларираме указател ptrom mun int*и му присвояваме адреса на първия елемент от numbersмасива. Забележете, че не е необходимо да използваме оператора адрес на (&), когато присвояваме адреса на указателя; самото име на масива действа като указател към неговия първи елемент.
		Тъй като указателят съдържа адреса на паметта на променлива, можем да използваме аритметика на указателя за достъп до елементите на масив. Например, за достъп до втория елемент от numbersмасива с помощта на показалеца ptr, можем да направим следното:
		int secondNumber = *(ptr + 1); // Accessing the second element using pointer arithmetic
		В този код използваме оператора за дереференция (*) за достъп до стойността, съхранена в мястото на паметта, към което сочи ptr + 1. Тук ptr + 1изчислява адреса на втория елемент чрез добавяне на подходящото отместване към указателя.
		Сега нека проучим как масивите могат да се предават като аргументи и параметри на функции. Когато предаваме масив на функция, ние всъщност предаваме указател към първия елемент на масива. Това означава, че всички модификации, направени в масива във функцията, ще засегнат оригиналния масив в извикващия код.
		Нека разгледаме един прост пример:
	Ex_01	void printArray(int arr[], int size) {
		for (int i = 0; i < size; i++) {
		printf("%d ", arr[i]);
		}
		printf("\n");
		}
		int main() {
		int numbers[5] = {10, 20, 30, 40, 50};
		printArray(numbers, 5);
		return 0;

		}
		В този код ние дефинираме функция printArray, която приема масив arr и неговия размер като параметри. Вътре във функцията използваме цикъл, за да обхождаме елементите на масива и да ги отпечатваме. След това във main функцията декларираме масив numbers и го предаваме заедно с размера му на printArray.
		Когато извикваме printArray(numbers, 5), ние предаваме масива numbers като аргумент. Не е необходимо обаче да използваме оператора адрес на (&), защото самото име на масива действа като указател към първия елемент. Това ни позволява директен достъп и манипулиране на елементите на масива във printArrayфункцията.
		Връзката между указатели и масиви ни предоставя мощен механизъм за ефективна работа с големи количества данни. Позволява ни да предаваме масиви на функции, без да правим копия, спестявайки памет и подобрявайки производителността.
		Това приключва нашия урок за указатели и масиви в C. Научихме как масивите и указателите са тясно свързани и как масивите могат да бъдат предавани като аргументи и параметри на функции. Отделете малко време, за да експериментирате с указатели и масиви, тъй като те са ключови понятия в програмирането на C. Благодаря ви за вниманието и ще се видим в следващия ни урок, където ще изследваме по-напреднали теми!
	3	**Въведение:** Добро утро/следобед на всички! Днес се гмуркаме по-дълбоко в света на програмирането на C и изследваме разширени операции с масиви. В този урок ще се съсредоточим върху **определянето на размера на масив и внедряването на алгоритми за обръщане, размяна, пресяване и други върху съдържанието на масив**. Тези техники ще ви помогнат да манипулирате ефективно масиви, отваряйки нови възможности за вашите програми. И така, да започваме!
		Определяне на размера на масив: Когато работите с масиви, е изключително важно да знаете техния размер. Размерът на масива представлява броя на елементите, които може да побере. В C можем да определим размера на масив с помощта на sizeofоператора. Нека да разгледаме един пример:
	Ex_02	#include <stdio.h>
		int main() {

```
int arr[ ] = {1, 2, 3, 4, 5};
int size = sizeof(arr) / sizeof(arr[0]);
printf("The size of the array is: %d\n", size);
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме масив arr с пет елемента.

За да определим размера на масива, използваме sizeof(arr), което връща общия брой байтове, заети от масива.

Въпреки това, за да получим действителния брой елементи, ние разделяме sizeof(arr) на sizeof(arr[0]), което ни дава размера на един елемент.

Накрая отпечатваме размера с помощта printf на оператор.

Обръщане на съдържанието на масив: Понякога трябва да обърнем реда на елементите в масив. Обръщането на масив може да бъде полезно в различни сценарии, като обработка на данни в обратен ред или прилагане на алгоритми като сортиране. Нека да видим как можем да обърнем съдържанието на масив:

Ex_03

```
#include <stdio.h>
void invertArray(int arr[ ], int size) {
int start = 0;
int end = size - 1;
while (start < end) {
int temp = arr[start];
arr[start] = arr[end];
arr[end] = temp;
start++;
end--;
}
}
```

```
int main() {
int arr[] = {1, 2, 3, 4, 5};
int size = sizeof(arr) / sizeof(arr[0]);
printf("The norma array is: ");
invertArray(arr, size);
printf("The inverted array is: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция invertArray, която приема масив arr и неговия размер като параметри.

Използваме два указателя start и end, за да итерираме от началото и края на масива към средата.

В рамките на while цикъла разменяме елементите при start и end с помощта на временна променлива.

След всяка размяна ние увеличаваме start и намаляваме end.

Накрая отпечатваме обърнатия масив с помощта на for цикъл.

Размяна на елементи в масив: Размяната на елементи в масив е основна операция, която може да се използва в различни алгоритми. Нека видим как можем да разменяме елементи при два дадени индекса в рамките на масив:

Ex_04

```
#include <stdio.h>
void swapElements(int arr[], int index1, int index2) {
int temp = arr[index1];
arr[index1] = arr[index2];
arr[index2] = temp;
}
int main() {
```

```
int arr[] = {1, 2, 3, 4, 5};
int size = sizeof(arr) / sizeof(arr[0]);
swapElements(arr, 1, 3);
printf("The array after swapping is: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция swapElements, която приема масив arr и два индекса index1u index2, като параметри.

Вътре във функцията използваме временна променлива, temp за да съхраним стойността в index1.

След това присвояваме стойността на index2 на arr[index1] и накрая присвояваме темрна arr[index2], ефективно разменяйки елементите.

Във mainфункцията извикваме swapElements( ) с индекси 1 и 3, за да разменим елементите на тези позиции.

Накрая отпечатваме модифицирания масив с помощта на for-цикъл.

Преместване на елементи в масив: Преместването на масив включва **преместване** на неговите елементи **наляво** или **надясно** с определен брой позиции. Пресяването може да бъде полезно, когато искаме да завъртаме елементи, да премахваме дубликати или да извършваме различни трансформации на данни. Нека да видим пример за преместване на елементи наляво:

Ex_05

```
#include <stdio.h>
void leftShift(int arr[], int size, int positions) {
for (int i = 0; i < positions; i++) {
int temp = arr[0];
for (int j = 0; j < size - 1; j++) {
arr[j] = arr[j + 1];
```

```
}
arr[size - 1] = temp;
}
}
int main() {
int arr[] = {1, 2, 3, 4, 5};
int size = sizeof(arr) / sizeof(arr[0]);
leftShift(arr, size, 2);
printf("The array after left shift is: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция leftShift, която приема масив arr, неговия размер и броя на позициите за преместване като параметри.

Използваме два вложени цикъла, за да изместим елементите наляво. Външният цикъл се изпълнява positionsвремена, показващи броя смени.

Във вътрешния цикъл изместваме всеки елемент наляво, като присвояваме стойността на arr[j + 1]на arr[j].

Накрая присвояваме оригиналната стойност на първия елемент на arr[size - 1], завършвайки лявото изместване.

Във mainфункцията извикваме leftShiftс positionsравно на 2, за да преместим елементите наляво два пъти.

Накрая отпечатваме модифицирания масив с помощта на forцикъл.

Заключение:

		Поздравления! Научихте напреднали операции с масиви в C програмирането. Обхванахме определянето на размера на масив с помощта на sizeof, обръщане на съдържанието на масив, размяна на елементи в масив и пресяване на елементи отляво. Като овладеете тези техники, можете да манипулирате ефективно масиви и да се справяте с по-сложни предизвикателства при програмирането. Продължавайте да практикувате и да изследвате силата на масивите във вашите програми. Приятно кодиране!
	4	Урок 04, част 04 : **Операции с масиви в C** програмиране - **основни алгоритми за търсене и подреждане на масиви**
		Въведение: Добро утро/следобед на всички! Днес продължаваме нашето пътуване из света на програмирането на C. В този урок ще разгледаме някои **основни алгоритми за търсене** и подреждане на масиви. Тези техники са жизненоважни при работа с по-големи масиви от данни, тъй като ни позволяват ефективно да търсим конкретни елементи и да подреждаме масива в желан ред. Така че, нека се потопим и открием тези мощни алгоритми!
		Алгоритми за търсене на масив: Търсенето на конкретни елементи в рамките на масив е често срещана задача в програмирането. Той ни позволява да намираме стойности, да определяме дали даден елемент съществува или да извличаме информация въз основа на специфични критерии. Нека разгледаме два популярни алгоритъма за търсене на масиви: линейно търсене и двоично търсене.
		a. Линейно търсене: Линейното търсене е ясен алгоритъм, който проверява всеки елемент в масива, докато намери съвпадение или стигне до края. Нека да разгледаме един пример:
	Ex_06	#include <stdio.h>
		int linearSearch(int arr[], int size, int key) {
		for (int i = 0; i < size; i++) {
		if (arr[i] == key) {
		return i;
		}
		}
		return -1; // Key not found

```
int main() {
int arr[] = {5, 8, 2, 10, 3};
int size = sizeof(arr) / sizeof(arr[0]);
int key = 10;
int result = linearSearch(arr, size, key);
if (result == -1) {
printf("Element not found\n");
} else {
printf("Element found at index: %d\n", result);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция linearSearch, която приема масив arr, неговия размер и ключа за търсене като параметри.

В рамките на функцията ние итерираме през всеки елемент от масива, използвайки цикъл for.

Ако текущият елемент съвпада с ключа, връщаме индекса на този елемент.

Ако не бъде намерено съвпадение след цикъла, ние връщаме -1, за да покажем, че ключът не е намерен.

Във mainфункцията извикваме linearSearchмасив с неговия размер и ключа за търсене.

Накрая отпечатваме дали елементът е намерен или не въз основа на върнатата стойност на linearSearch.

b. Двоично търсене: Двоичното търсене е ефективен алгоритъм за търсене, който изисква масивът да бъде сортиран. Той многократно разделя пространството за търсене наполовина, докато се намери желаният елемент. Да видим пример:

Ex_07

```
#include <stdio.h>
int binarySearch(int arr[], int low, int high, int key) {
while (low <= high) {
```

```
int mid = low + (high - low) / 2;
if (arr[mid] == key) {
return mid;
}
if (arr[mid] < key) {
low = mid + 1;
} else {
high = mid - 1;
}
}


int main() {
int arr[] = {2, 3, 5, 8, 10};
int size = sizeof(arr) / sizeof(arr[0]);
int key = 8;
int result = binarySearch(arr, 0, size - 1, key);
if (result == -1) {
printf("Element not found\n");
} else {
printf("Element found at index: %d\n", result);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция binarySearch, която приема масив arr, ниските и високите индекси на пространството за търсене и ключа за търсене като параметри.

Използваме whileцикъл, за да разделяме многократно пространството за търсене, докато lowстане по-голямо от high.

Във всяка итерация изчисляваме средния индекс (mid) като (low + high) / 2.

		Ако средният елемент съвпада с ключа, връщаме неговия индекс.
		Ако средният елемент е по-малък от ключа, коригираме low индекса на mid + 1.
		Ако средният елемент е по-голям от ключа, коригираме high индекса на mid - 1.
		Ако не бъде намерено съвпадение след цикъла, ние връщаме -1, за да покажем, че ключът не е намерен.
		Във mainфункцията извикваме binarySearch със сортиран масив, ниския и високия индекси и ключа за търсене.
		Накрая отпечатваме дали елементът е намерен или не въз основа на върнатата стойност на binarySearch.
		Алгоритми за подреждане на масиви: Подреждането на масив ни позволява да сортираме елементите му в определен ред, като възходящ или низходящ. Сортирането е ключова операция в програмирането и служи като основа за много алгоритми. Нека проучим два общи алгоритъма за подреждане на масиви: сортиране по избор и сортиране с балончета.
		a. Сортиране чрез избиране /Selection Sort/ : Сортирането при избор многократно избира най-малкия елемент от несортираната част на масива и го разменя с елемента в началото на сортираната част. Да видим пример:
	Ex_08	#include <stdio.h>
		void selectionSort(int arr[], int size) {
		for (int i = 0; i < size - 1; i++) {
		int minIndex = i;
		for (int j = i + 1; j < size; j++) {
		if (arr[j] < arr[minIndex]) {
		minIndex = j;
		}
		}
		if (minIndex != i) {
		int temp = arr[i];
		arr[i] = arr[minIndex];
		arr[minIndex] = temp;
		}

```
}
}
int main() {
int arr[] = {5, 2, 8, 3, 10};
int size = sizeof(arr) / sizeof(arr[0]);
selectionSort(arr, size);
printf("The sorted array is: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция selectionSort( ), която приема масив arru неговия размер като параметри.

Използваме два вложени цикъла. Външният цикъл итерира от началото на масива до предпоследния елемент.

В рамките на външния цикъл приемаме текущия елемент за минимален и съхраняваме неговия индекс в minIndex.

Вътрешният цикъл започва от следващия елемент и го сравнява с предполагаемия минимум. Ако бъде намерен по-малък елемент, актуализираме minIndex.

След като намерим минималния елемент, ние го разменяме с елемента в началото на сортираната част (m.e. arr[i]).

Повтаряме този процес, докато целият масив бъде сортиран.

Във mainфункцията се обаждаме selectionSortс масив и неговия размер.

Накрая отпечатваме сортирания масив с помощта на forцикъл.

b. Сортиране с "мехурче" /BubbleSort/ : Сортирането с "мехурче" многократно сравнява съседни елементи и ги разменя, ако са в грешен ред, като постепенно премества по-големите елементи към края. Да видим пример:

Ex_09

```c
#include <stdio.h>
void bubbleSort(int arr[], int size) {
for (int i = 0; i < size - 1; i++) {
for (int j = 0; j < size - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
}
int main() {
int arr[] = {5, 2, 8, 3, 10};
int size = sizeof(arr) / sizeof(arr[0]);
bubbleSort(arr, size);
printf("The sorted array is: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция bubbleSort, която приема масив arr и неговия размер като параметри.

Използваме два вложени цикъла. Външният цикъл итерира от началото на масива до предпоследния елемент.

Вътрешният цикъл сравнява съседни елементи и ги разменя, ако са в грешен ред.

		След всяка итерация на вътрешния цикъл най-големият елемент в несортираната част се придвижва към края.
		Повтаряме този процес, докато целият масив бъде сортиран.
		Във main функцията се обаждаме bubbleSort( ) с масив и неговия размер.
		Накрая отпечатваме сортирания масив с помощта на for-цикъл.
		Заключение:
		Поздравления! Научихте някои фундаментални алгоритми за търсене на масиви и подреждане на масиви в C програмирането. Изследвахме линейното търсене и двоичното търсене за ефективно намиране на елементи в масив. Разгледахме също сортиране по избор и сортиране с мехурчета за подреждане на елементи в масив. Като овладеете тези алгоритми, можете да решите широк кръг от проблеми, включващи масиви. Продължавайте да практикувате и да изследвате света на програмирането. Приятно кодиране!
	5	Урок 04: Част 05 : **Разширени операции с масиви** в C програмиране - **масиви от индекси, масиви от указатели и масиви от масиви.**
		Въведение: Добро утро/следобед на всички! Връщаме се с още един интересен урок по програмиране на C. В тази сесия ще разгледаме някои разширени концепции, свързани с масивите. Ще се потопим в **масиви от индекси, масиви от указатели и масиви от масиви.** Тези концепции предоставят мощни начини за ефективно манипулиране и организиране на данни. Така че, нека се задълбочим в тези теми и да отключим нови възможности с масивите!
		Масиви от индекси: Масивът от индекси е масив, който съхранява индексите на елементи от друг масив. Това ни позволява да създадем специфичен ред или картографиране за елементите. Нека видим пример за използване на масив от индекси за сортиране на друг масив:
	Ex_10	#include <stdio.h>
		void sortArray(int arr[], int size, int indexArr[]) {
		for (int i = 0; i < size - 1; i++) {
		for (int j = i + 1; j < size; j++) {
		if (arr[indexArr[i]] > arr[indexArr[j]]) {
		int temp = indexArr[i];

```
indexArr[i] = indexArr[j];
indexArr[j] = temp;
}
}
}
}
int main() {
int arr[] = {5, 2, 8, 3, 10};
int size = sizeof(arr) / sizeof(arr[0]);
int indexArr[] = {0, 1, 2, 3, 4};
sortArray(arr, size, indexArr);
printf("The sorted array is: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[indexArr[i]]);
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние дефинираме функция sortArray, която приема масив arr, неговия размер и масив от индекси indexArrкато параметри.

Използваме два вложени цикъла за сравняване и размяна на елементи въз основа на индексите, съхранени в indexArr.

Сравнението и размяната се извършват върху действителните елементи на масива, като се използват индексите от indexArr.

Във mainфункцията декларираме масив indexArrсъс същия размер като arr. Първоначалните стойности представляват индексите във възходящ ред.

Извикваме sortArrayфункцията за сортиране arrвъз основа на индексите в indexArr.

Накрая отпечатваме сортирания масив, като използваме индексите от indexArrза достъп до елементите в правилния ред.

		Масиви от указатели: Масив от указатели е масив, в който всеки елемент е указател към друг тип данни. Той осигурява гъвкавост при управление на паметта и ни позволява да създаваме динамично масиви с различни размери. Нека да видим пример за масив от указатели към низове:
	Ex_11	#include <stdio.h>
		int main() {
		char* names[] = {"Alice", "Bob", "Charlie", "David"};
		int size = sizeof(names) / sizeof(names[0]);
		printf("The names are: ");
		for (int i = 0; i < size; i++) {
		printf("%s ", names[i]);
		}
		return 0;
		}
		Обяснение:
		В този кодов фрагмент ние декларираме масив names, който съдържа указатели към низове.
		Всеки елемент от namese указател към низов литерал, представляващ името на човек.
		Размерът на масива namesce определя с помощта на sizeofоператора.
		Във mainфункцията отпечатваме имената чрез достъп до всеки елемент от namesмасива с помощта на forцикъл.
		Използваме %sспецификатора на формата, за да отпечатаме низовете, към които сочи всеки елемент.
		Масиви от масиви (многомерни масиви): Масивите от масиви, известни също като многомерни масиви, ни позволяват да съхраняваме данни в таблична форма. Те предоставят начин за представяне на матрици, таблици или всякакви структури от данни с множество измерения. Нека да видим пример за двуизмерен масив, представляващ матрица:
		#include <stdio.h>
		int main() {
		int matrix[3][3] = {{1, 2, 3},
		{4, 5, 6},

```
{7, 8, 9}};
printf("The matrix is:\n");
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
printf("%d ", matrix[i][j]);
}
printf("\n");
}
return 0;
}
```

Обяснение:

В този кодов фрагмент ние декларираме двуизмерен масив matrixс три реда и три колони.
Всеки елемент от matrixпредставлява цяло число.
Във mainфункцията използваме вложени forцикли, за да обикаляме редовете и колоните на матрицата.
Отпечатваме всеки елемент с помощта на индексите matrix[i][j].
След отпечатването на всеки ред вмъкваме знак за нов ред, за да форматираме изхода.

Заключение:

Поздравления! Разгледахте усъвършенствани концепции, свързани с масиви в C програмирането. Покрихме масиви от индекси, които ни позволяват да създадем специфичен ред или картографиране за елементи. Обсъдихме и масиви от указатели, които осигуряват гъвкавост при управление на паметта и работа с масиви с динамичен размер. И накрая, изследвахме масиви от масиви, които ни позволяват да представяме данни в таблична форма или с множество измерения. Като разберете и използвате тези концепции, вие можете ефективно да управлявате сложни структури от данни и алгоритми. Продължавайте да практикувате и да откривате силата на масивите във вашето програмиране. Приятно кодиране!

6 Урок 04 част 06 : **Разширени операции с масиви** в C програмиране - **създаването на динамичен масив**

		Въведение: Добро утро/следобед на всички! Добре дошли отново в нашия урок по програмиране на C. В тази сесия ще изследваме **създаването на динамичен масив**, мощна техника, която ни позволява да създаваме масиви с различни размери по време на изпълнение. Динамичните масиви осигуряват гъвкавост при работа с непредвидими данни или когато размерът на масива трябва да се променя динамично. И така, нека се потопим в света на динамичните масиви и да видим как те могат да подобрят нашите програми!
		Създаване на динамичен масив: В C ние обикновено определяме размера на масива по време на компилация. Има обаче сценарии, при които трябва да създадем масиви с различни размери по време на изпълнение. Тук влиза в действие създаването на динамичен масив. В C динамичните масиви се създават с помощта на указатели и функции за разпределяне на памет като malloc( ) и calloc( ). Нека разгледаме пример за динамично създаване на масив :
		#include <stdio.h>
		#include <stdlib.h>
		int main() {
		int size;
		int* dynamicArray;
		printf("Enter the size of the array: ");
		scanf("%d", &size);
		dynamicArray = (int*)malloc(size * sizeof(int));
		if (dynamicArray == NULL) {
		printf("Memory allocation failed\n");
		return 1;
		}
		printf("Enter %d elements:\n", size);
		for (int i = 0; i < size; i++) {
		scanf("%d", &dynamicArray[i]);
		}
		printf("The elements of the array are: ");
		for (int i = 0; i < size; i++) {
		printf("%d ", dynamicArray[i]);
		}

```
free(dynamicArray);
return 0;
}
```

Обяснение:

В този кодов фрагмент ние декларираме указател dynamicArray от тип int за задържане на динамично създадения масив.

Ние подканваме потребителя да въведе размера на масива с помощта на scanf( ).

След това разпределяме памет за динамичния масив с помощта на malloc( ) функцията, умножавайки размера по, sizeof(int) за да разпределим подходящия брой байтове.

Проверяваме дали разпределението на паметта е било успешно, като проверяваме дали dynamicArraye NULL.

Ако разпределението на паметта е неуспешно, отпечатваме съобщение за грешка и се връщаме от програмата.

След това подканваме потребителя да въведе елементите на масива с помощта на цикъл и scanf( ).

След въвеждането на елементите ги отпечатваме с друг цикъл.

Накрая освобождаваме паметта, разпределена за динамичния масив, като използваме free( ) функцията, за да избегнем изтичане на памет.

Динамичен двуизмерен масив: Създаването на динамичен масив може да се приложи и към многоизмерни масиви. Нека разгледаме пример за динамично създаване на двуизмерен масив:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int rows, cols;
int** dynamicMatrix;
printf("Enter the number of rows: ");
scanf("%d", &rows);
printf("Enter the number of columns: ");
scanf("%d", &cols);
dynamicMatrix = (int**)malloc(rows * sizeof(int*));
```

```
if (dynamicMatrix == NULL) {
printf("Memory allocation failed\n");
return 1;
}
for (int i = 0; i < rows; i++) {
dynamicMatrix[i] = (int*)malloc(cols * sizeof(int));
if (dynamicMatrix[i] == NULL) {
printf("Memory allocation failed\n");
return 1;
}
}
printf("Enter the elements of the matrix:\n");
for (int i = 0; i < rows; i++) {
for (int j = 0; j < cols; j++) {
scanf("%d", &dynamicMatrix[i][j]);
}
}
printf("The matrix is:\n");
for (int i = 0; i < rows; i++) {
for (int j = 0; j < cols; j++) {
printf("%d ", dynamicMatrix[i][j]);
}
printf("\n");
}
for (int i = 0; i < rows; i++) {
free(dynamicMatrix[i]);
}
free(dynamicMatrix);
return 0;
}
```

		Обяснение:
		В този кодов фрагмент ние декларираме указател dynamicMatrixот тип int** за задържане на динамично създадения двуизмерен масив.
		Ние подканваме потребителя да въведе броя на редовете и колоните за матрицата с помощта на scanf( ).
		Разпределяме памет за редовете на матрицата с помощта на malloc() функцията, умножавайки броя на редовете по, за sizeof(int*) да разпределим подходящия брой байтове.
		След това итерираме всеки ред и разпределяме памет за колоните с помощта на malloc(), умножавайки броя на колоните по, за sizeof(int)да разпределим подходящия брой байтове.
		Проверяваме дали разпределението на паметта за всеки ред и колона е било успешно.
		Ако разпределението на паметта е неуспешно, отпечатваме съобщение за грешка и се връщаме от програмата.
		След това подканваме потребителя да въведе елементите на матрицата с помощта на вложени цикли и scanf().
		След като въведем елементите, отпечатваме матрицата с помощта на вложени цикли.
		Накрая освобождаваме паметта, разпределена за динамичната матрица, като първо освобождаваме паметта за всеки ред и след това освобождаваме паметта за самите редове.
		Заключение:
		Поздравления! Научихте силата на създаването на динамичен масив в C програмирането. Изследвахме динамично създаване на едномерни и двумерни масиви с помощта на указатели и функции за разпределяне на памет като malloc(). Динамичните масиви ни позволяват да обработваме различни размери на масиви по време на изпълнение, осигурявайки гъвкавост при управлението на данни. Не забравяйте да освободите разпределената памет, като използвате, free()за да избегнете изтичане на памет. Динамичните масиви откриват нови възможности за вашите програми, позволявайки ви да обработвате ефективно непредвидими данни. Продължавайте да практикувате и да изследвате света на динамичните масиви. Приятно кодиране!