Масиви

Бърз heads-up: променлива vs. масив.

- Променлива – int number = 3;
  Представлява начин, по който запазваме някаква стойност (3), давайки ѝ име (number), за да може програмата да се обръща към стойността по-късно.
  Можем да си мислим за променливата като за чекмедже с етикет „number" отвън и стойността 3 вътре и всеки път, когато погледнем какво има в чекмеждето, откриваме 3.
  Но може да ни се наложи да съхраняваме няколко стойности в една и съща променлива. Тогава не можем да направим просто int number = 3, 5; т.к. това ще даде компилационна грешка. За това има измислен специален механизъм, а именно:
- Масив – int numbers[2] = {3, 5};
  За масива можем да си мислим като за скрин с много чекмеджета – всяко си има етикет numbers[i] отвън и някаква стойност отвътре.

Как се създава масив?

- **<тип> <име_на_масива>[<размер_на_масива>] { = <редица_от_константни_изрази>} опц,** където:
  <*тип*> е типът на елементите, от които се състои масивът;
  <*име_на_масива*> е идентификатор (име, което си избираме за нашата променлива масив);
  <*размер_на_масива*> е константно цяло число и определя колко елементи ще има масивът;
  <*редица_от_константни_изрази*> е опционална; тя задава последователно стойности на елементите на масива; ако нямаме такава редица, масивът е неинициализиран.

Пр.:
```
int arr[5] = {1, 2, 3, 4, 5};
char arr[3] = {'a', 'b', 'c'};
```

Как работи това? - *какво се случва, когато напишем:* `int arr[5] = {1, 2, 3, 4, 5};` ?
В паметта на програмата се заделят 5 последователни „кутийки" (адреса) при компилация. Размерът (във B) на кутийките е голям, колкото е голям типът на променливите, които пазим в масива (за int масив една кутийка ще бъде 4B). Масивът всъщност е просто последователност от елементи в паметта, на която последователност предварително сме ѝ задали размера (чрез размера на масива).

Как НЕ се създава масив? - *не можем да направим следното*:
```
int n;
cin >> n;
int arr[n];
```

Защо? Защото размерът на масива трябва да е ясен при компилиране на програмата. В случая, когато компилираме, знаем само, че стойността на n е цяла и ще се въведе от потребителя. Не знаем дали потребителят ще въведе 1 или 100000 => компилаторът не знае колко памет да задели за масива ни.

- Ако все пак ни трябва да имаме масив от n елемента, какво правим?
  Заделяме повече памет с константен размер, проверяваме дали n < от заделената памет и ако да, си вършим работата с масива.

Пр.:
```
int n;
cin >> n;
int arr[100];
if(n < 100) {
        <каквото_искаме_да_правим_с_масива>;
} else {
        cout << "Not enough memory allocated!" << endl;
}
```

Индексация на елементите в масива:
Номерацията (индексацията) на елементите на масива започва от 0.
Тоест, ако искаме да достъпим стойността на първия елемент, ще трябва да достъпим стойността на нулевия елемент: cout << numbers[0]; извежда 3.

- *N.B. За да достъпим i-тия елемент (отляво надясно) на масива arr, използваме arr[i-1].*

Пр.:

```
int arr[5] = {1, 2, 3, 4, 5};
cout << arr[0]; //1
cout << arr[1]; //2
cout << arr[2]; //3
cout << arr[3]; //4
cout << arr[4]; //5
```

```
char arr[3] = {'a', 'b', 'c'};
cout << arr[0]; //'a'
cout << arr[1]; //'b'
cout << arr[2]; //'c'
```

- *N.B. За да достъпим i-тия елемент (отдясно наляво) на масива arr, който има размер n, използваме arr[n - 1 - i].*

Как се задават стойности на елементите на масива? – имаме 3 възможности:

- Задаваме стойности на елементите на масива *при създаването* на масива:

```
int arr[5] = {1, 2, 3, 4, 5};
cout << arr[0]; //1
cout << arr[1]; //2
cout << arr[2]; //3
cout << arr[3]; //4
cout << arr[4]; //5
```

- Не задаваме стойности на елементите на масива при създаването му, а ги *въвеждаме от конзолата*:

```
int arr[5];
cin >> arr[0]; //ако въведем 3, arr[0] става 3;
cin >> arr[1]; //ако въведем 17 000, arr[1] става 17 000 и т.н.;
cin >> arr[2];
cin >> arr[3];
cin >> arr[4];
```

- Не задаваме стойности на елементите на масива при създаването му, но им *присвояваме стойности на други променливи или константни стойности:*

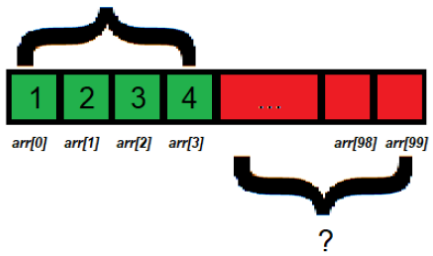
```
int arr[5];
arr[0] = 3; //3
arr[1] = 4; //4
int b = 6;
arr[2] = 6; //6
//arr[3], arr[4] остават недефинирани
```

Обхождане на масив (*Обхождане на масив == преминаване през всичките му елементи точно по веднъж)*
Обхождането се моделира лесно с for-цикъл. На i-тата итерация на for-цикъла достъпваме i-тия елемент на масива.
Тук става ясно защо много често, когато използваме for-цикли започваме да броим от 0 и продължаваме до n-1 включително, вместо да броим от 1 и да продължаваме до n включително – защото сме свикнали индексацията на масива да става от 0.
Пр.: Да се въдедат n цели числа (n <= 100) и след като са въведени, да се изведат.

```
int n;
cin >> n;
int arr[100];
for (int i = 0; i < n; i++) {
        cin >> arr[i];
}
for (int i = 0; i < n; i++) {
        cout << arr[i] << ", "
}
```

Но след като сме заделили константна памет и тя е по-голяма от размера n на масива, как разбираме до къде да обходим паметта? Просто *обхождаме масива до там, до където знаем, че има елементи.*
За горния пример в паметта имаме 100 кутийки, които са част от масива arr, но от тях сме инициализирали само n на брой; нека сме въвели n = 4. Тогава, ако обходим до повече от 4 ще достъпим памет, в която нямаме дефинирани стойностите на елементите на масива. Затова винаги обхождаме само до n.

n = 4;



Какво означава променлива (в случая елемент на масива) да остане недефиниран/а?
Означава, че променливата няма да има стойност и всеки път, когато се достъпва, поведението ѝ ще бъде недефинирано (може да изкарва странни символи, неочаквани стойности, да не изкарва нищо и т.н.)
Затова, винаги си дефинирайте променливите (елементите на масива).