Масивите ни улесняват живота

Подобно на функциите, масивите са конструкция, без която всъщност бихме могли да постигнем същия резултат, с малко упоритост. Те са в някакъв смисъл просто последователност от променливи от един и същи тип. Защо тогава просто не използваме променливи от един и същи тип, вместо тях?

Краткият отговор е “защото понякога тези променливи стават много”
Да вземем за пример един много опростен списък с печалбите на някакъв магазин (например за банички), за които за всеки ден от една седмица знаем по колко пари нетно е направил, като се разберем, че ще бъдат в цели числа. Например:

```
#include <iostream>
using namespace std;

int main () {
	int d1;
	int d2;
	int d3;
	int d4;
	int d5;
	int d6;
	int d7;

	cin>> d1;
	cin>> d2;
	cin>> d3;
	cin>> d4;
	cin>> d5;
	cin>> d6;
	cin>> d7;
}
```

Ами ако трябваше да са записите за година? Много са. Можем да ги запишем един по един, но това е лошо решение.
Класическото решение с масив:

```
#include <iostream>
using namespace std;

int main () {
	int d[365];
	for(int i=0; i< 365; ++i) {
		cin>> d[i];
	}
}
```

Не всички променливи трябва да живеят в масиви обаче - ако имаме 3-4 независими променливи, няма смисъл да ги слагаме в масив. Да разгледаме например сумата на две числа:

1) С масив:
```
int numbers[3];
cin>> numbers[0]>> numbers[1];
numbers[2] = numbers[0] + numbers[1]
```
2) Без масив:
```
int number1, number2, sum;
cin>> number1>> number2;
sum = number1 + number2;
```

Вариант 2) очевидно е по-четим. Това е понеже сме имали свободата да кръстим всяко отделно “нещо” с името му: имаме първо число, второ число и сума.

Масивите ни помагат да се ориентираме, когато имаме “група” от подобни елементи, към която искаме да се обръщаме с едно име. Нещо като множество - както в примера с дните. Като обаче имаме въведен ред - в масива си има първи елемент, втори елемент и така нататък - има наредба.

Тази наредба идва от представянето на масива в паметта: знаем, функцията си има стекова рамка, в която живее паметта за променливите, които са декларирани в нея. От тази памет, част е за нашия масив. В нея са наредени един след друг елементите на масива и съответно на всеки му трябва памет, колкото трябва на една променлива от този тип (да си припомним декларацията на масив - <име> <тип> [размер]). Заради това компилатора знае: 0вият елемент се намира на 0 byte-а разстояние от началото на масива, 1вия елемент се намира на колкото е byte-а типа надясно от него, 2рия на два пъти повече и т.н. Ефективно е същото като да дефинираме променливите отделно - само тогава нямаме гаранция за наредбата в паметта (което не ни вълнува). Тази адресация на самите елементи е и причината индексът на масива да е целочислен тип (arr[5]).
Сега, понеже го има този индекс в декларацията на масива, някой може да си помисли, че в него може да има произволен тип елементи - тоест например да въведем n на вход и да декларираме масива с този размер и той да стане с този размер. Това обаче няма да се случи, понеже (да си припомним) дефинирането на масив с n елемента не е по-различно от дефинирането на n променливи. Компилаторът иска да знае колко памет ще заемат те, преди да се изпълни програмата, за да знае колко голяма ще е стековата рамка - ако масивът е 10 елемента ще е различно от ако е 10000, например. Затова размерът на масива трябва да е константа или литерал в декларацията му.
Началните стойности на елементите няма да са нещо смислено - дефинирането на масив с n елемента = дефинирането на n променливи - те са “недефинирани”.
Имаме няколко опции за това как да го напълним:

1) Можем да го напълним по нормалния начин с присвоявания
2) Можем да го инициализираме с литерал от вида {x0, x1, … xl}, l< n

```
int arr[] = {1,2,3,4,5}; // [1,2,3,4,5]
Int arr[10] = {}; // 0и
Int arr[10] = {1,2,3,4,5} // [1,2,3,4,5,0,0,0,0,0]
```

Sizeof на масив връща броят на byte-овете, които заема - тоест ако искаме да видим колко елементи има трябва да вземем sizeof(arr)/sizeof(int)
Ако пишем извън масива е лошо - не знаем нищо за тази памет – тя може да принадлежи на друга програма, например и да ни е забранено да я пипаме – в този случай програмата ще прекъсне и ще ни даде грешка. Ако имаме право да достъпваме паметта, ще вземем каквото има на нея, но има голяма вероятност то да не е нещо смислено и почти 100% вероятност да не е каквото ни трябва