Pointer arithmetic

- За разлика от референциите, както вече казахме пойнтърите могат да променят адреса, към който сочат
- На създателите на езика им е хрумнала идеята, че може да има няколко съседни клетки от един и същи тип една до друга
- Тъй като пойнтърът знае към какъв тип променлива сочи, той знае точно колко байта да се измести в паметта (наляво или надясно), за да стигне до следващата или по-следващата клетка
- Така се появяват и тъй наречените Pointer arithmetic, които позволяват извършване на аритметични операции с пойнтъри

Размер на примитивните типове данни

*Информацията е валидна за x86 VC компилатор

- 1 byte – bool, char
- 2 bytes – short (int)
- 4 bytes – int, long, float, enum
- 8 bytes – long long, double, long double
- Размерът на пойнтъра съвпада с този на типа на променливата
  (при x64 VC компилатор пойнтърите са двойно по-големи)

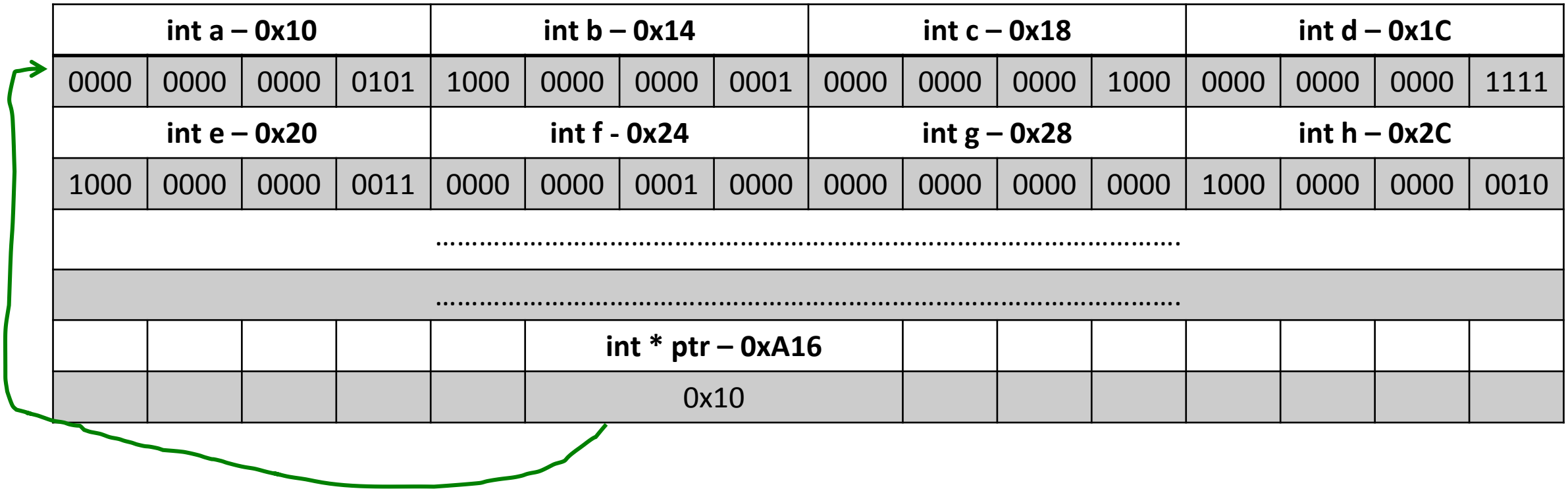
Pointer arithmetic

- Да кажем, че магически сме накарали компилатора да сложи 8 променливи от тип int една след друга в паметта, като:
  - a = 5
  - b = -1
  - d = 15
  - e = -3
  - f = 16
  - g = 0
  - h = -2

int a				**int b**				**int c**				**Int d**			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e				**int f**				**int g**				**int h**			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010

Pointer arithmetic

- Нека да създадем пойнтър int* ptr, който сочи към a.



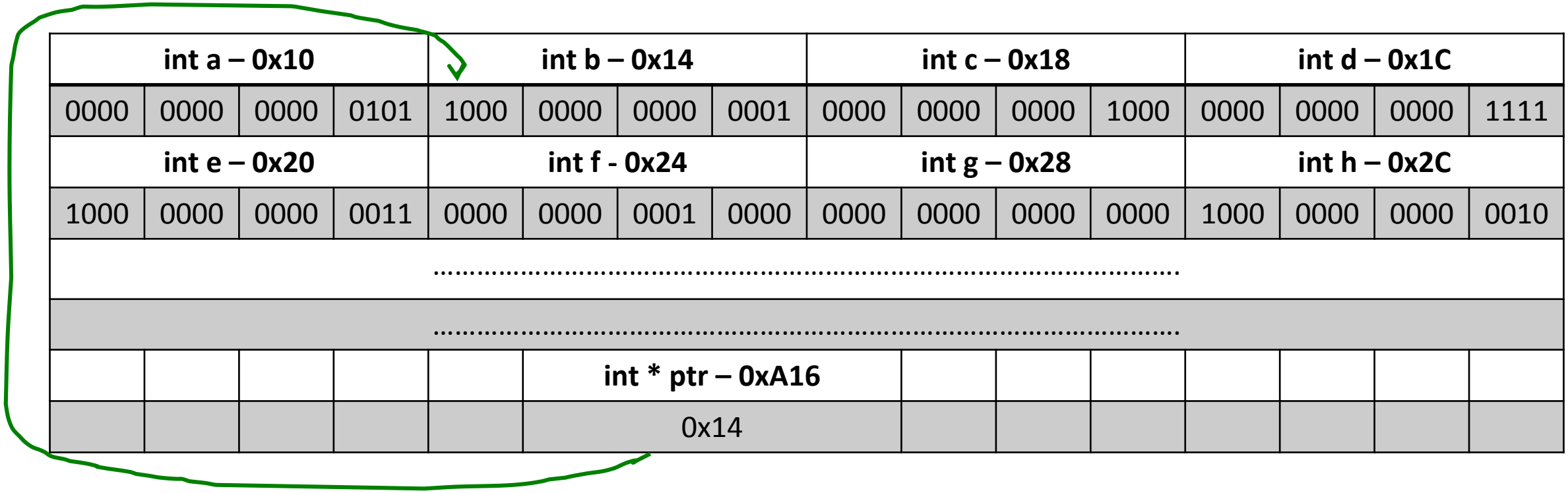
Pointer arithmetic

- Тъй като ptr знае, че сочи към int, то за да го накараме да сочи към съседния(тоест следващите 4 клетки), трябва да му кажем премести се с 1 клетка: ptr+1

int a – 0x10				**int b – 0x14**				**int c – 0x18**				**int d – 0x1C**			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				**int f - 0x24**				**int g – 0x28**				**int h – 0x2C**			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
…….															
…….															
					int * ptr – 0xA16										
					0x10										

Pointer arithmetic

- Можем и тотално да сменим адреса, към който сочи ptr: ptr+=1

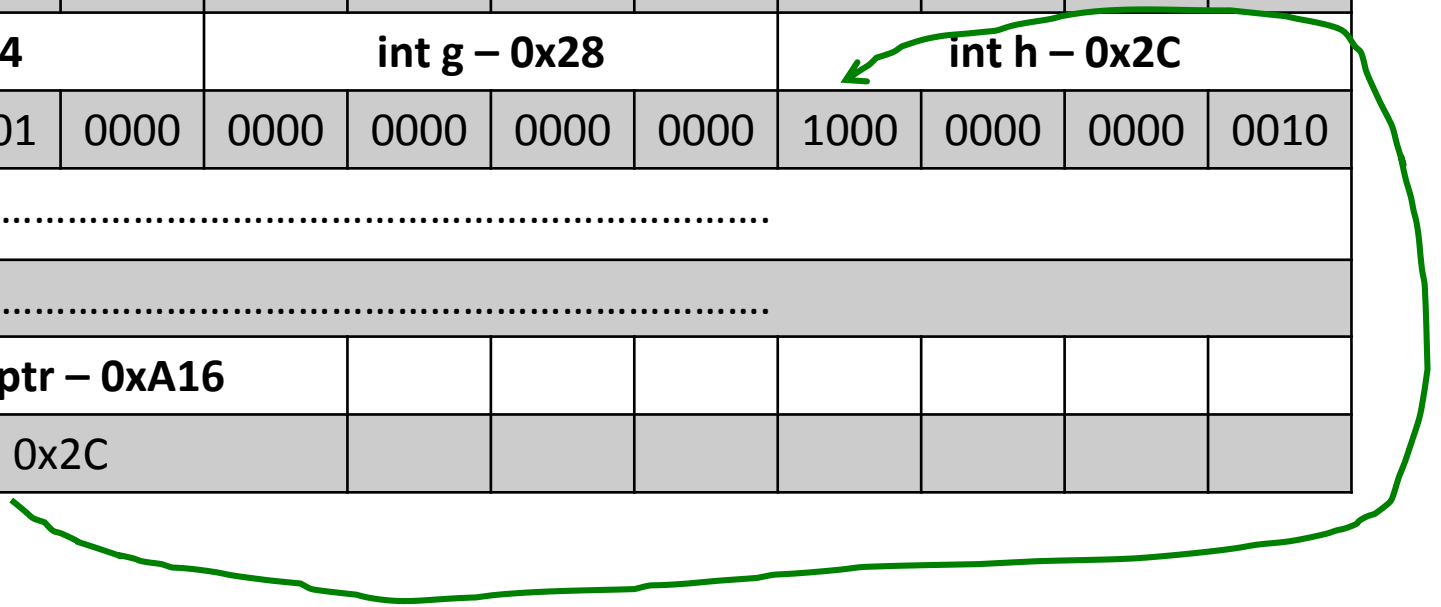


int a – 0x10				**int b – 0x14**				**int c – 0x18**				**int d – 0x1C**			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				**int f - 0x24**				**int g – 0x28**				**int h – 0x2C**			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
…….															
…….															
					int * ptr – 0xA16										
					0x14										

Pointer arithmetic

- Можем да действаме и по-смело ptr +=6

int a – 0x10				**int b – 0x14**				**int c – 0x18**				**int d – 0x1C**			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				**int f - 0x24**				**int g – 0x28**				**int h – 0x2C**			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
…….															
…….															
					int * ptr – 0xA16										
					0x2C										



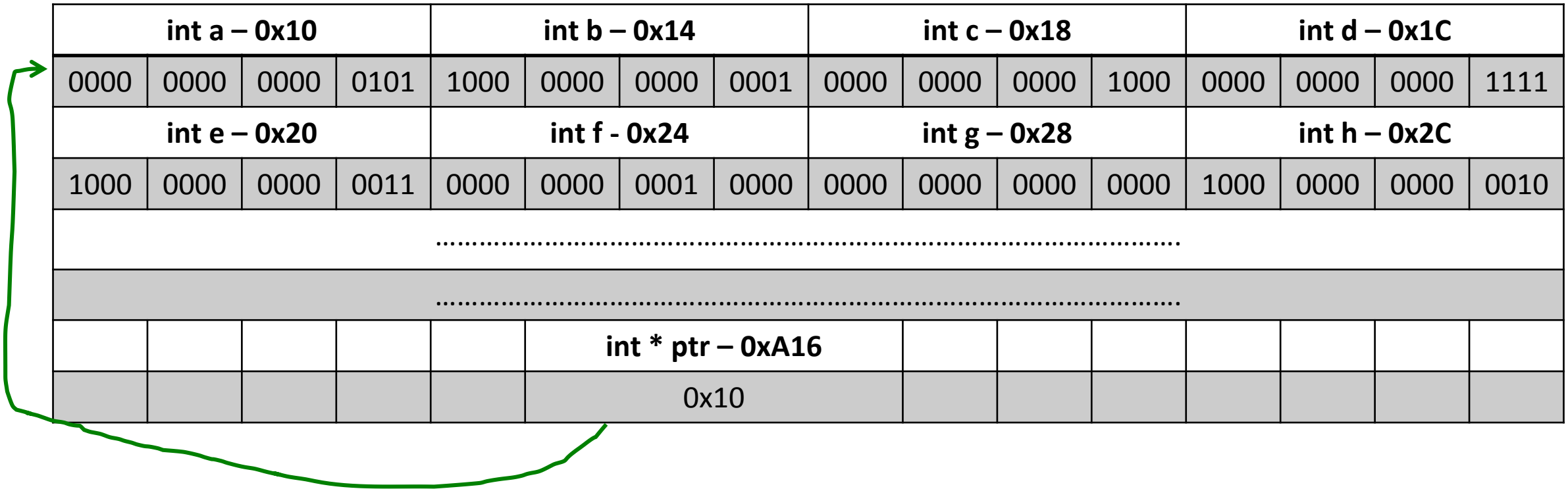
Pointer arithmetic

- Разбира се, можем и да се връщаме назад: --ptr

int a – 0x10				**int b – 0x14**				**int c – 0x18**				**int d – 0x1C**			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				**int f - 0x24**				**int g – 0x28**				**int h – 0x2C**			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
…….															
…….															
					int * ptr – 0xA16										
					0x28										

Pointer arithmetic

- Да се върнем в началото: ptr -= 6



Pointer arithmetic

- За да обходим всичките клетки и да изпишем какво съдържат, можем да напишем следния код:

```
for(unsigned i = 0; i<8;++i)
{
        std::cout<<*(ptr + i)<<‘ ‘;
}
```

- Изглежда ли ви познато?

Масиви

- Масивът е съставен тип данни
- Представя крайна редица от еднотипни елементи
- Позволява произволен достъп до всеки негов елемент по номер (индекс)

Синтаксис

- <тип> <идентификатор> [[<константа]] [= { <константа> [, <константа>] }] ;
- Примери:
  - bool b[10];
  - const int y = 3;
  - double x[y] = { 0.5, 1.5, 2.5 }, z = 3.8; //z е просто double
  - int a[] = { 3 + 2, 2 * 4 }; ⇐⇒ int a[2] = { 5, 8 };

- За всички фенове на Java и C#

bool[10] b; е невалиден израз

Операции за работа с масив

- Достъп до елемент по индекс: <масив>[<цяло_число>]

- Примери:
  - x = a[2]; (rvalue)
  - a[i] = 7; (lvalue)

- Броенето на индексите започва от 0

- Оператор sizeof() връща разликата на първата и последната клетка

Операции за работа с масив

- За масив от тип int, който съдържа 5 елемента, sizeof() ще ви върне 20, защото в масива има 20 клетки (5*4)

- За да вземете големината на целия масив, трябва да използвате следната хитринка: sizeof(<име>)/sizeof(<тип>)

  sizeof(<име>)/sizeof(<име>[<число>])

- Внимание: няма проверка за коректност на индекса, затова използвайте 0!

Операции за работа с масив

- Няма присвояване a = b
- Няма поелементно сравнение a == b винаги връща false ако a и b са различни масиви, дори и да имат еднакви елементи
- Няма операции за вход и изход std::cin >> a; std::cout << a;
- std::cout << a; извежда адреса на a (не важи за символен низ)

Операции за работа с масив

- Не можете да подадете масив като параметър на функция

- Можете да имате като параметър:
  - <име>[]
  - <име>[<число>]
  - пойнтър

- Който и от трите метода да изберете, ще получите едно и също, но само ако подавате пойнтър може да сте спокойни, че всичко, което не смятате да променяте, ще е const

Пример

```
void foo(const int arr [])
{
        int *p = nullptr;
        arr = p;
}
int main()
{
        int a[5];
        a[2] = 5;
        foo(a);
        std::cout << 2[a];    //защо това извежда 5?
        return 0;
}
```

Пример

```
void foo(const int * const arr )
{
        int *p = nullptr;
        arr = p;    //това вече не се компилира
}
int main()
{
        int a[5];
        a[2] = 5;
        foo(a);
        std::cout << 2[a];
        return 0;
}
```

Операции за работа с масив

- Не можете да направите функция, която връща масив
- Можете да направите функция, която връща пойнтър към първия елемент на масив
- Работата с масиви в C++ понякога е сложна и ограничаваща, затова в стандартната библиотека има много различни имплементации, които много улесняват програмиста
- За да ги оцените подобаващо и да ги разберете, трябва първо да се поизцапате в калта

Масиви и пойнтъри

- Може да се каже, че масивът е по-специален константен пойнтър

- Реално масивът е пойнтър към 0вия елемент в поредицата, но има и по-специални свойства като:
  - запазване на последователни блокове памет при инициализация
  - не просто сочи към 0вия елемент, а към цялата редица
  - знание колко му е размера
- Операцията [<число>] работи и за пойнтъри и е равносилна на *(<пойнтър/име_на_масив> + число)

- Demo4 и Demo5

Двумерни масиви

- Така както пойнтър може да сочи към пойнтър, то може да има и масив от масиви

- Аналогията не е случайна, защото int ** има същата логика като int [][]

- int a [5][4] - масив, който съдържа 5 масива, които съдържат 4 елемента от тип int (все едно имаме масив от пойнтъри)

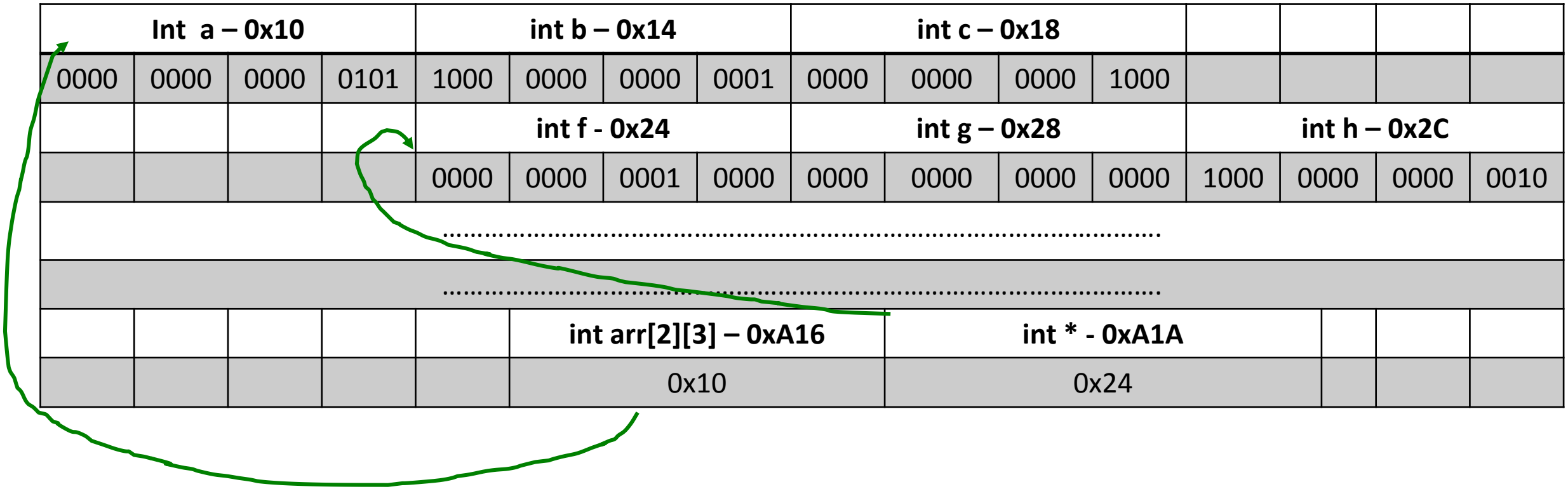
Двумерни масиви

- Също както при обикновените масиви, щом имаме масив от пойнтъри, то те са един до друг в паметта, но не е нужно това, към което сочат, също да е последователно в паметта

- Demo6

Двумерен масив

- Нека видим примерно представяне на двумерен масив int arr[2][3] (два масива с по 3 елемента/два поинтъра към int)



Подаване на двумерен масив като параметър на функция

- Има 2 основни начина за подаване на двумерен масив, като и двата има преимущества и недостатъци

1. Подаване като масив с [][<число>]
   - изисква да се знае колко мерни са масивите в масива предварително
   - масивите в масива имат свойствата на истински масив
2. Подаване като двоен пойнтър **
   - няма ограничение относно големината на масивите в масива
   - губят се свойствата на масива

Многомерни масиви

- Защо да спираме само с двумерни масиви?

- Реално можем да имаме n-мерни масиви
  - Двумерните масиви представляват някаква таблица - Demo7
  - Тримерните масиви са като някакъв паралелепипед
  - N-мерните – илюминати

- Подаването на многомерни масиви като формални параметри може да е tricky, затова избягвайте да го правите