Maciej Sobieraj

Lecture 5

Outline

1. Aggregating data into arrays
    1. **Sorting data: in real life and in the computer memory**
    2. Initiators - the simple way to set an array
    3. Not only ints
    4. Pointers: another kind of data in the “C” language
    5. Pointers vs. arrays: different forms of the same phenomenon
    6. The string: a very special vector
2. Quiz

Sorting an array

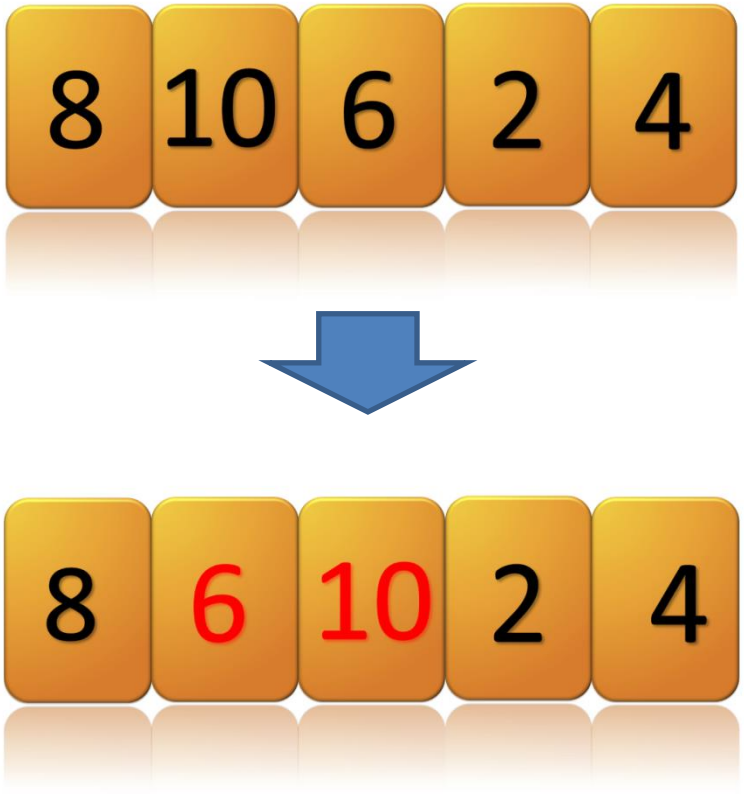
- The array can be sorted in two ways:
  - **increasing** (or more precisely – **non-decreasing**) if, in every pair of adjacent elements, the former element is not greater than the latter;
  - **decreasing** (or more precisely – **non-increasing**) if, in every pair of adjacent elements, the former element is not less than the latter.

Sorting an array

- We'll try to use the following approach:
  - we'll take the first and second elements and compare them; if we determine that they're in the **wrong order** (the first is greater than the second), we'll **swap** them around;
  - if they're in the right order, we'll do nothing.
  - A glance at our table confirms the second condition – the elements #1 and #2 are in the proper order, as *8 < 10*.

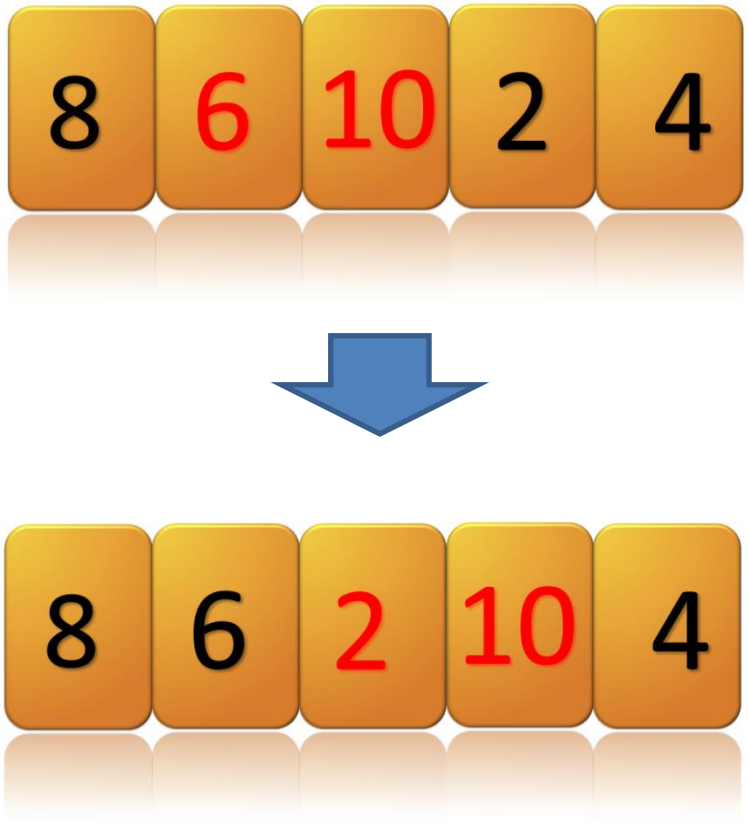
Sorting an array

- We can go further and look at the third and fourth elements.



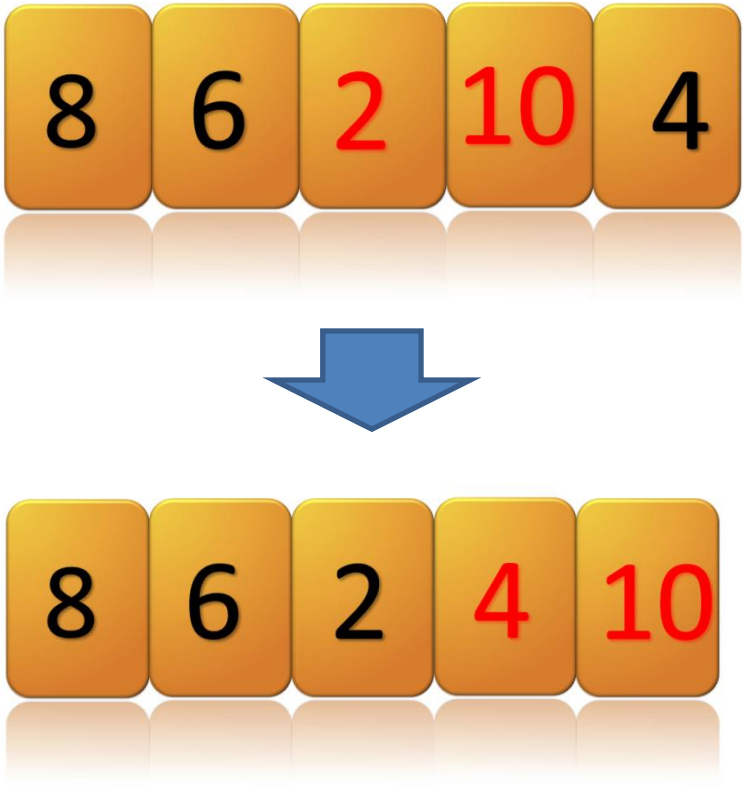
Sorting an array

- Now we check the fourth and fifth elements.



Sorting an array

- The first pass through the array is complete. We're still far from finishing our job



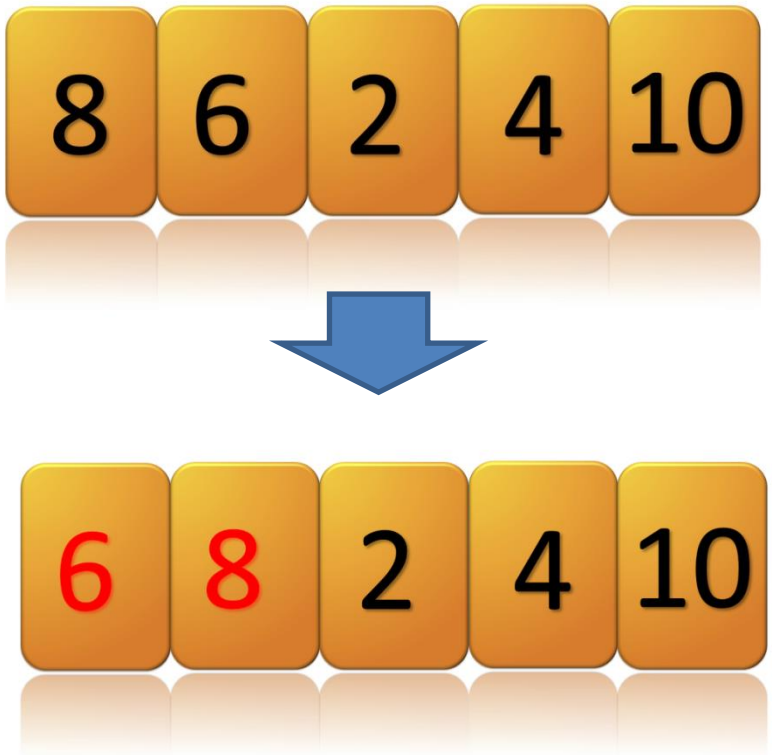
Sorting an array

- Now, for a moment, try to imagine this array in a slightly different way

- The sorting method derives its name from this same observation – it's called a **bubble sort**.

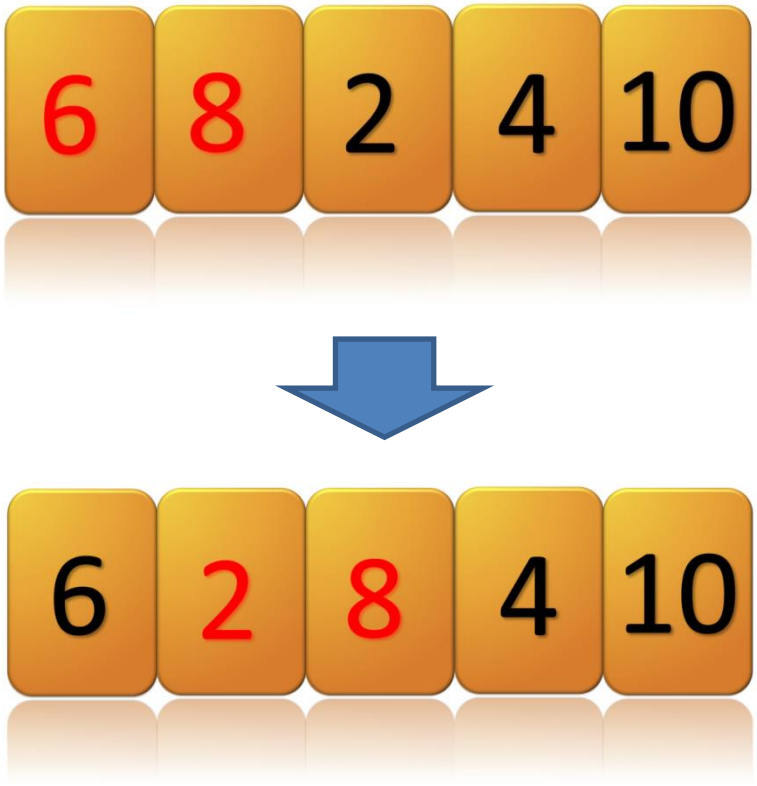
Sorting an array

- We start with the second pass through the array. We look at the first and second elements - a swap is necessary!



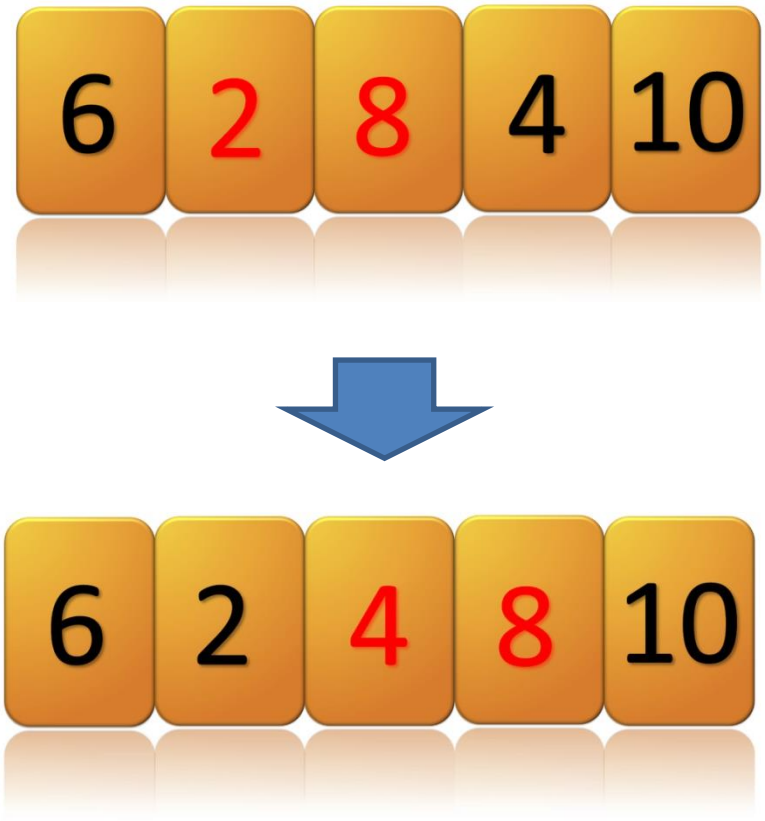
Sorting an array

- Now the second and third elements: yep, *8* is a bubble and goes up to the surface



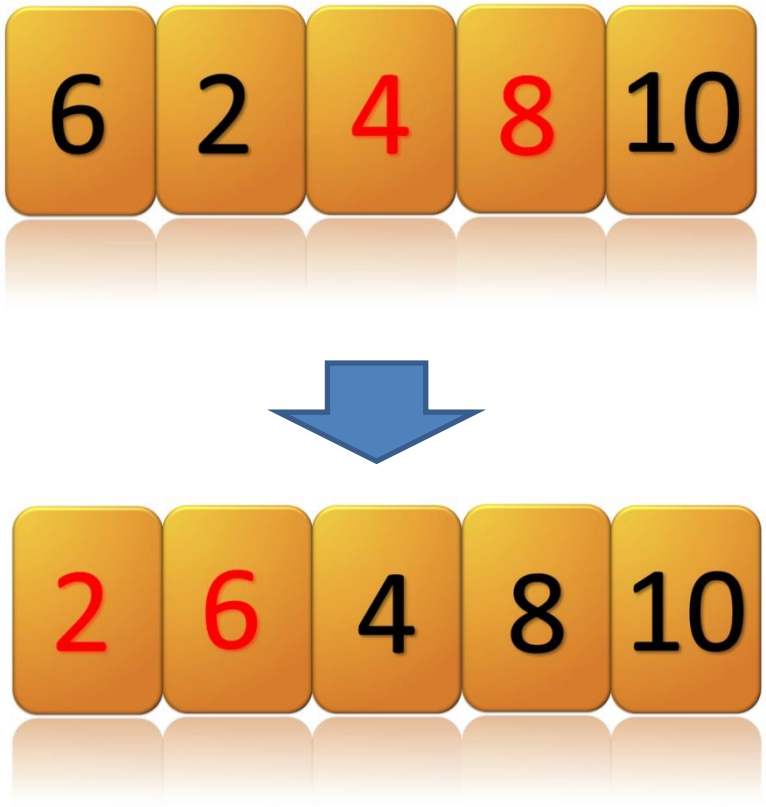
Sorting an array

- Time for the third and fourth elements: we have to swap them too



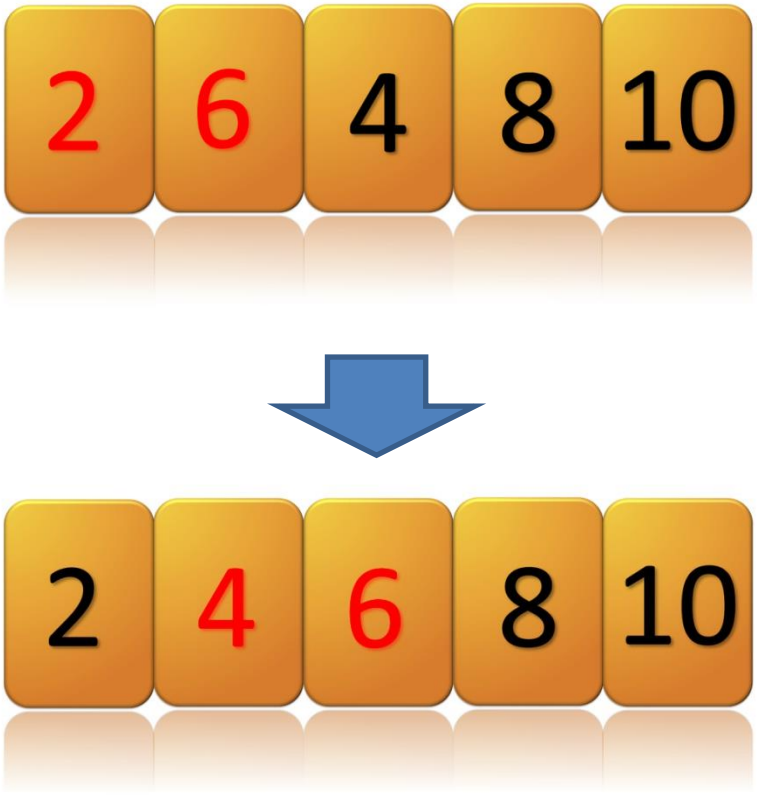
Sorting an array

- The second pass is finished and *8* is already in place. We start the next pass immediately.



Sorting an array

- Now *6* wants to find its place. We'll help it and swap the second and third elements.



Sorting an array

```
int numbers[5]; /* array to be sorted */
int i;  /* a variable for the loop */
int aux; /* auxiliary variable for swaps */

/* we need 5 – 1 comparisons – why? */
for(i = 0; i < 4; i++) {
   /* compare adjacent elements */

   if( numbers[i] > numbers[i + 1]) {
      /* if we went here it means that we have to swap the elements */
     aux = numbers[i];
     numbers[i] = numbers[i + 1];
     numbers[i + 1] = aux;
   }
```

Sorting an array

- How many passes do we need to sort the entire array?
  - We answer this by doing the following: we introduce another variable; its task is to observe if any swap was done during the pass or not; if there was no swap, then the array is already sorted and nothing more has to be done.
  - We declare a variable named *swapped* and we assign a value of *0* to it to indicate that there were no swaps. Otherwise, it will be *assigned 1*.

Sorting an array

```
int numbers[5];
int i, aux;
int swapped;

do {  /* we will decide if we need to continue this loop */
   swapped = 0;  /* no swap occured yet */

   for(i = 0; i < 4; i++)
      if(numbers[i] > numbers[i + 1]) {
         swapped = 1;
         aux = numbers[i];
         numbers[i] = numbers[i + 1];
         numbers[i + 1] = aux;
      }
} while(swapped);
```

Sorting an array

```c
#include <stdio.h>
int main(void) {
    int numbers[5];
    int i, aux;
    int swapped;
    /* ask the user to enter 5 values */
    for(i = 0; i < 5; i++) {
        printf("\nEnter value #%i\n",i + 1);
        scanf("%d",&numbers[i]);
    }
    /* sort them */
    do {
        swapped = 0;
        for(i = 0; i < 4; i++) {
            if(numbers[i] > numbers[i + 1]) {
                swapped = 1;
                aux = numbers[i];
                numbers[i] = numbers[i + 1];
                numbers[i + 1] = aux;
            }
        }
    } while(swapped);
    /* print results */
    printf("\nSorted array: ");
    for(i = 0; i < 5; i++)
        printf("%d ",numbers[i]);
    printf("\n");
    return 0;
}
```

Outline

1. Aggregating data into arrays
    1. Sorting data: in real life and in the computer memory
    2. **Initiators - the simple way to set an array**
    3. Not only ints
    4. Pointers: another kind of data in the “C” language
    5. Pointers vs. arrays: different forms of the same phenomenon
    6. The string: a very special vector
2. Quiz

Initiators - the simple way to set an array

- The vector initiator is simply a **list of values enclosed inside curly brackets**.

```
int vector[5] = { 0,1,2,3,4 };
```

Initiators - the simple way to set an array

- If you provide fewer values than the size of an array, like this, nothing bad will happen. The compiler determines that those elements for which you did not specify any value should be **set to *0***.

```
int vector[5] = { 0,1,2 };
```

Initiators - the simple way to set an array

- If you provide more elements than can be stored in an array, like this, you'll get an error. Some old compilers can notify you without stopping compilation.
- This is called a **compilation warning**.

```
int vector[5] = { 0,1,2,3,4,5,6 };
```

Initiators - the simple way to set an array

- This is legal and will force the compiler to **assume that the size** of the array is the **same as the length** of the initiator.

```
int vector[] = { 0,1,2,3,4,5,6 };
```

- The *vector* array will be considered declared in the following way:
  - **int** vector[7] = { 0,1,2,3,4,5,6 };

Outline

1. Aggregating data into arrays
    1. Sorting data: in real life and in the computer memory
    2. Initiators - the simple way to set an array
    3. **Not only ints**
    4. Pointers: another kind of data in the “C” language
    5. Pointers vs. arrays: different forms of the same phenomenon
    6. The string: a very special vector
2. Quiz

Not only ints

- You can use **arrays of any other type**. For example – this is an array in which you can store 10 floating-point values.

```
float FloatArr[10];
```

Not only ints

- And you can store 20 characters here.
- The latter array will, however, be treated a little differently by the compiler. Its initiator will be different, too.

```
char surname[20];
```

Outline

1. Aggregating data into arrays
    1. Sorting data: in real life and in the computer memory
    2. Initiators - the simple way to set an array
    3. Not only ints
    4. **Pointers: another kind of data in the “C” language**
    5. Pointers vs. arrays: different forms of the same phenomenon
    6. The string: a very special vector
2. Quiz

Pointers – the absolute basics

- **Pointers** are also **values**, but are different from those we've operated with so far.
- Memory size is expressed in units called **bytes**, and you also know that when you declare any variable, for example, in such an obvious and simple way, the variable **occupies a little piece** of the computer memory.

```
int i;
```

Pointers – the absolute basics

- From now on, we're also interested in **where** this value is stored.
- This trait of the data (to say it more formally, **attribute**) is often called the **address**. We all live at certain addresses, just like every variable "lives" at its address too.
- Try to see this important difference:
  - the **value** of the variable is what the variable stores;
  - the **address** of the variable is information about where this variable is placed (where it *lives*).

Pointers – the absolute basics

- **Pointers** are used to store information about the location (address) of any other data. We can say that pointers are like **signposts**. They don’t say anything about the place itself, but they show clearly how to reach it.

The first pointer

- **The presence of the asterisk means that *p* is a pointer** and will be used to store information about the location of the data of type *int*.

```
int *p;
```

How do we assign a value?

- Can we assign a value to the pointer? Of course we can, in the same way you can assign any value to any other variable: by using the = operator.
- Using a **literal is not an option**.

```
p = 148324;
```

How do we assign a value?

- A pointer which is assigned a value of zero is called a **null pointer** (as in Latin, *nullus* – none).

```
p = 0;
```

How do we assign a value?

- The **NULL** symbol is actually equal to **zero**. It looks like a variable but you cannot modify its value.
- *NULL* should be assigned only to pointers.
- if you want to use the NULL symbol, you have to **include one of the following header files**: *stdio.h* or *stddef.h*.

```
p = NULL;
```

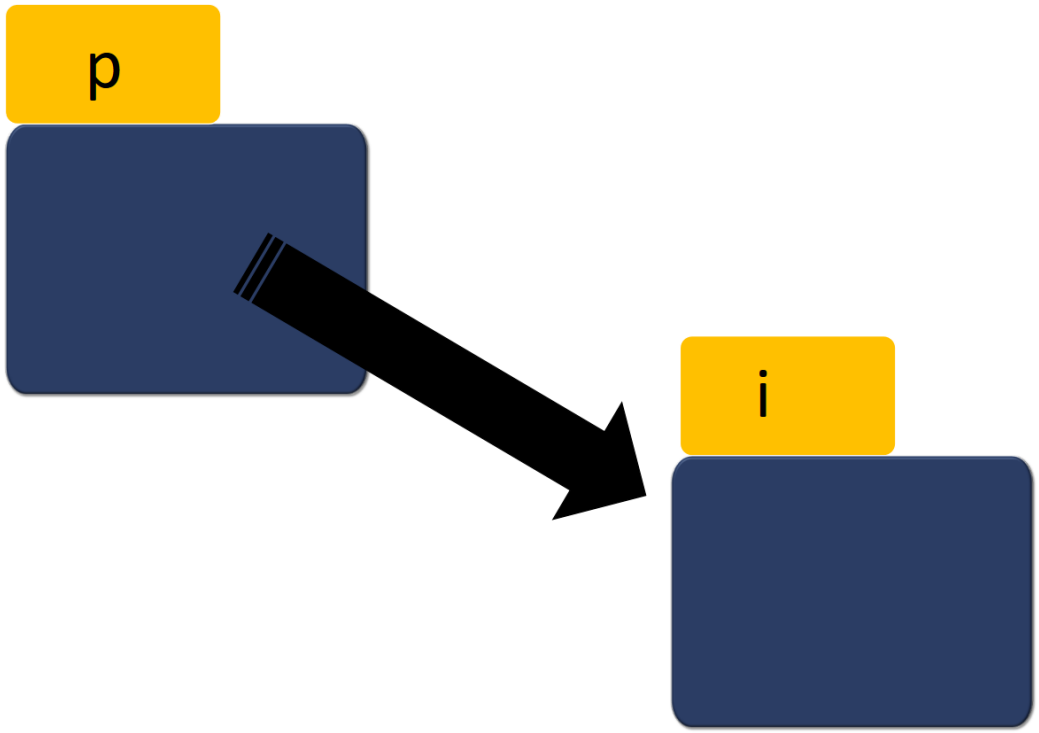
How do we assign a value?

- We may assign to the pointer a value which **points to any already existing variable**.
- To do that, we need an *&* operator, called the **reference operator**.

```
p = &i;
```

How do we assign a value?

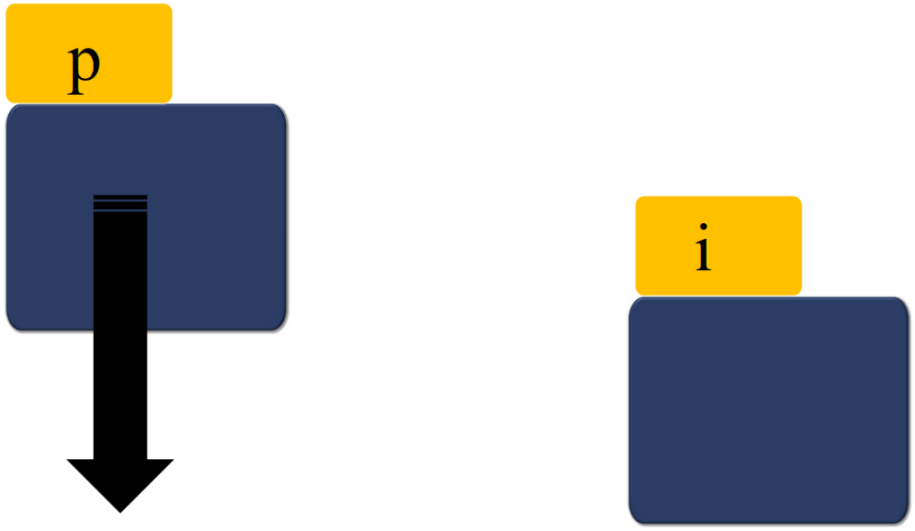
- After completing the assignment, the *p* variable will point to the place where the *i* variable is stored in the memory.



How do we assign a value?

- If you assign NULL to the pointer, it'll look like this. From now on, the *p* pointer points to neither the *i* variable nor to any other variable.

```
p = NULL;
```



How do we assign a value?

- We declare a variable of type *int* (*ivar*) and a variable of type *int* * (ptr)

```
int ivar, *ptr;
```

- Now let's assign the value of 2 to the *ivar* variable

```
ivar = 2;
```

How do we assign a value?

- Now we make the *ptr* pointer point to the *ivar* variable.

```
ptr = &ivar;
```

How do we assign a value?

- How do we get a value pointed to by the pointer?
- We have to use a well-known operator (the asterisk: “*”) but in a completely new way – as a **dereferencer**.

How do we assign a value?

- The following invocation will display 2 on the screen, as the *printf's* second argument is the **dereferenced** *ptr* **value**

```
printf("%d",*ptr);
```

How do we assign a value?

- If you write a statement like the one here → you **won't change the pointer value**. You'll instead change the value pointed to by the pointer.

```
*ptr = 4
```

How do we assign a value?

- Don't forget that if you declare a pointer in the following way:
  - ANY_TYPE *pointer;
- it means that:
  - the pointer variable is of type ANY_TYPE*
  - the * pointer expression is of type ANY_TYPE

Another new operator

- The new operator expects that its argument is a **literal**, or a **variable**, or an **expression** enclosed in parentheses, or the **type name**
- The operator provides information on **how many bytes of memory its argument occupies**

sizeof

Another new operator

```
int i; char c;

i = sizeof c;
```

- Variable *i* will be assigned the value of 1, because *char* values always occupy one byte.
- Note that we can achieve the same effect by writing:
  - **i** = **sizeof**(char);

Another new operator

```
char tab[10];

i = sizeof tab;
```

- Variable *i* will be set to the value of 10, because this is the number of bytes occupied by the **entire** *tab* array.

Another new operator

```
char tab[10];

i = sizeof tab[1];
```

- Variable *i* will be set to the value of 1

Another new operator

```
int i;

i = sizeof i;
```

- Values of the *int* type occupy 32 bits, i.e. 4 bytes in most modern compilers/computers, **but we cannot guarantee** that this is true in all cases.

Another new operator

```c
#include <stdio.h>

int main(void) {

   printf("This computing environment uses:\n");
   printf("%d byte for chars",sizeof(char));
   printf("%d bytes for shorts",sizeof(short int));
   printf("%d bytes for ints",sizeof(int));
   printf("%d bytes for longs",sizeof(long int));
   printf("%d bytes for long longs",sizeof(long long int));
   printf("%d bytes for floats",sizeof(float));
   printf("%d bytes for doubles",sizeof(double));
   printf("%d bytes for pointers",sizeof(int *));
   return 0;
}
```

Another new operator

! ~ (type) ++ -- + - * & sizeof	unary
* / %	
+ -	binary
<< >>	
< <= > >=	
== !=	
&	
|	
&&	
||	
= += -= *= /= %= &= ^= |= >>= <<=	

Outline

1. Aggregating data into arrays
    1. Sorting data: in real life and in the computer memory
    2. Initiators - the simple way to set an array
    3. Not only ints
    4. Pointers: another kind of data in the “C” language
    5. **Pointers vs. arrays: different forms of the same phenomenon**
    6. The string: a very special vector
2. Quiz

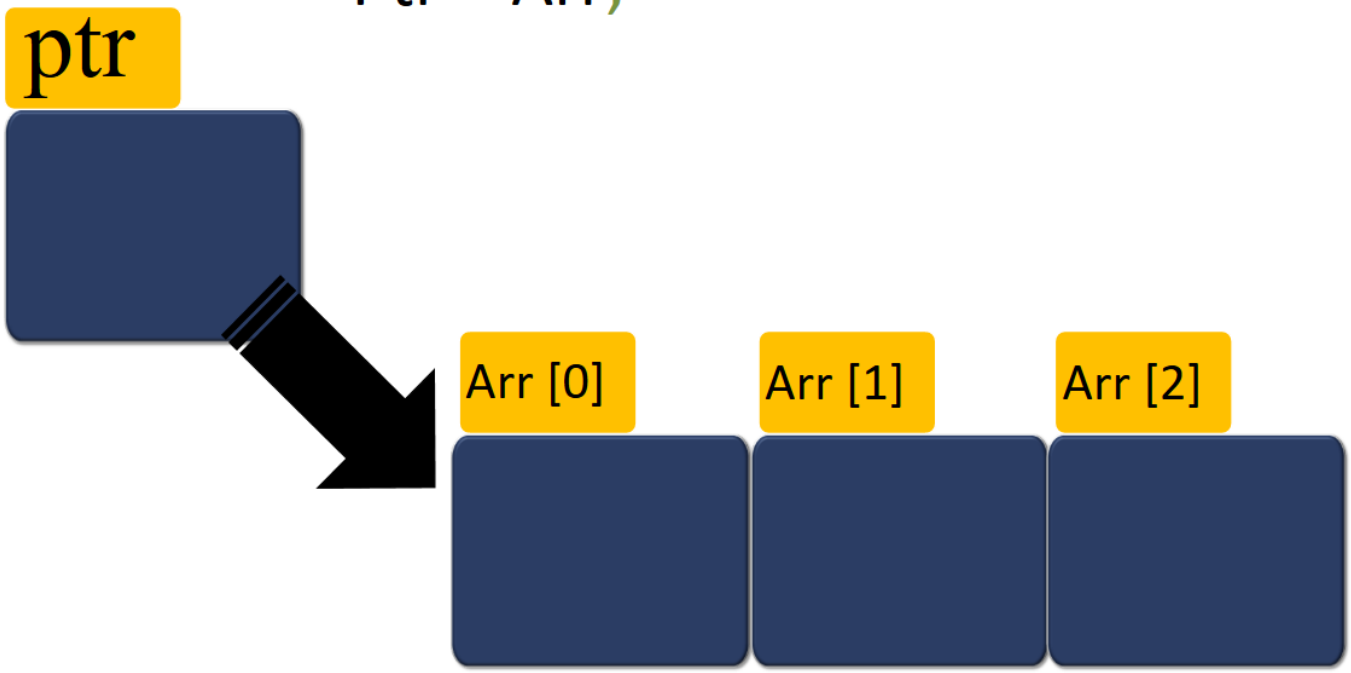
Pointers vs. arrays

- What do **pointers and arrays** have in common?
  - if you see the **name of an array** without the indices, then it's always a **synonym of the pointer pointing to the first element of the array**.

```
int *Ptr, Arr[3];
```

Pointers vs. arrays

```
int *Ptr, Arr[3];
Ptr = &Arr[0];
Ptr = Arr;
```



Pointers vs. arrays

- The two assignments that follow the declaration set *Ptr* to the same value. In other words, the following comparison is always true:
  - Arr == &Arr[0]

Pointers vs. arrays

- The arithmetic of pointers is significantly different from the arithmetic of integers, as it's relatively reduced and allows the following operations:
  - **adding an integer** value to a pointer, giving a pointer (*ptr* + *int* → *ptr*);
  - **subtracting an integer** value from a pointer, giving a pointer (*ptr* – *int* → *ptr*);
  - **subtracting a pointer from a pointer**, giving an integer (*ptr* – *ptr* → *int*);
  - **comparing the two pointers** for equality or inequality (this gives a value of type *int* of either *true* or *false*) (*ptr == ptr* → *int; ptr != ptr* → *int*).

Pointers vs. arrays

- At this point, *ptr1* points to the first element of *array*.

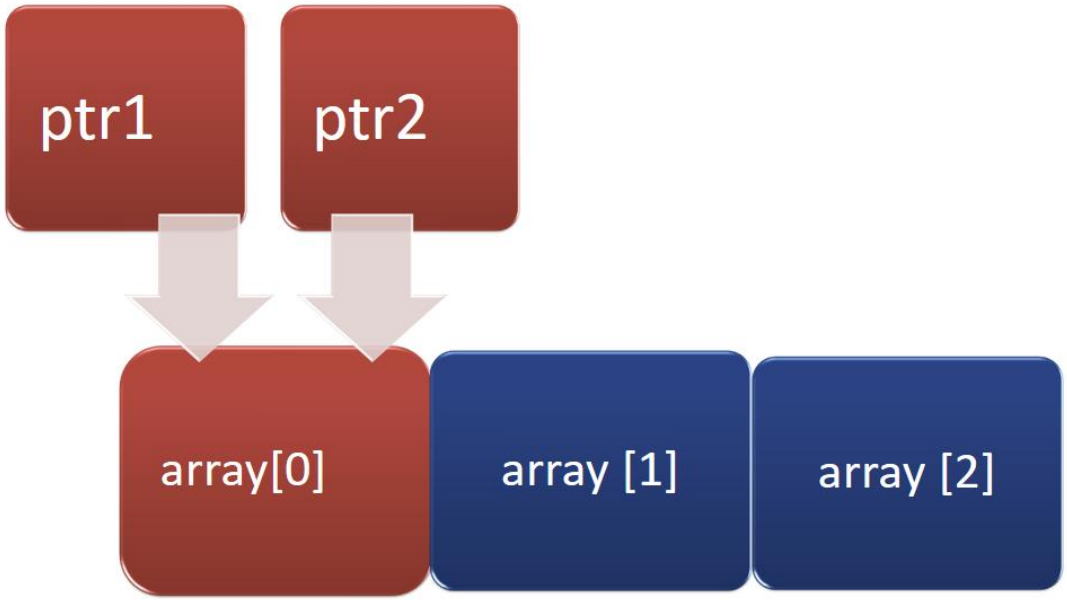
```
int *ptr1, *ptr2, array[3], i;

ptr1 = array;
```

Pointers vs. arrays

- After the following assignment, *ptr2* **points to the first element of** *array*, too

```
ptr2 = ptr1;
```



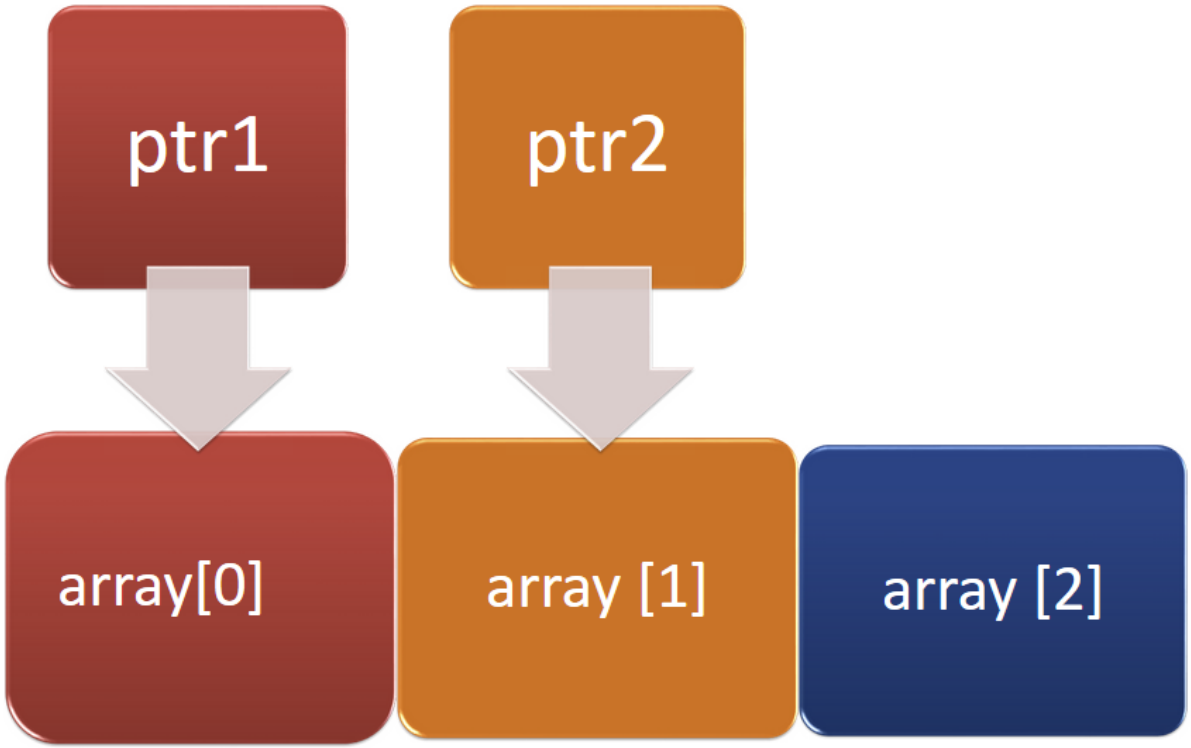
Pointers vs. arrays

- We can check if the two pointers are equal – yes, they are, as they point to the same element of the *array*.

```
if(ptr2 == ptr1) {
:
:
}
```

Pointers vs. arrays

```
ptr2++;
```



Pointers vs. arrays

```
ptr2 = ptr2 + 1;
```

- We can interpret this operation as follows:
  - it has taken into account what type is pointed to by the pointer – in our example it's *int*;
  - it has determined **how many bytes of memory the type occupies** (the *sizeof* operator is used automatically for that purpose) – in our case it's *sizeof (int)*;
  - the value we want to add to the pointer is multiplied by the given size;
  - the address which is stored in the pointer is **increased** by the resulting product.

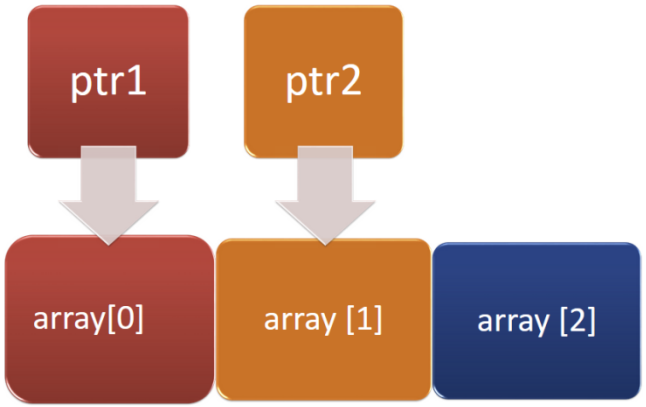
Pointers vs. arrays

- What would happen if we added *2* instead of *1*?
  - In this case the *ptr2* would be increased by (*2 * sizeof (int)*) and thus *ptr2* would move through **two** *int* values and would point to the third element of the array (namely, *array[2]*).
  - The comparison
    - ptr1 == ptr2
  - is obviously false, while this one
    - ptr1 != ptr2
  - is true, as the addresses the pointers point to differ.

Pointers vs. arrays

- The final result tells us how many variables of a given type (i.e. *int*) fit between the addresses stored in the pointers.

```
i = ptr2 - ptr1;
```



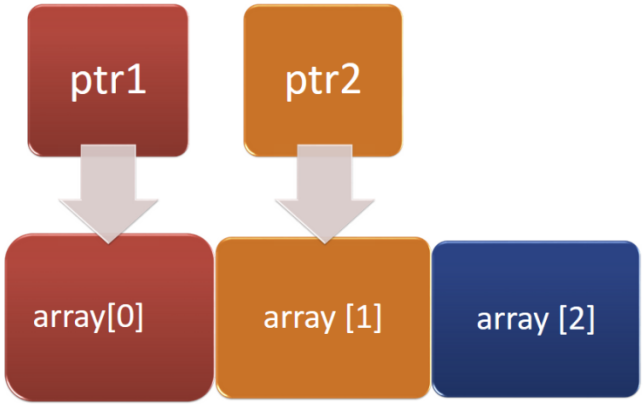
Pointers vs. arrays

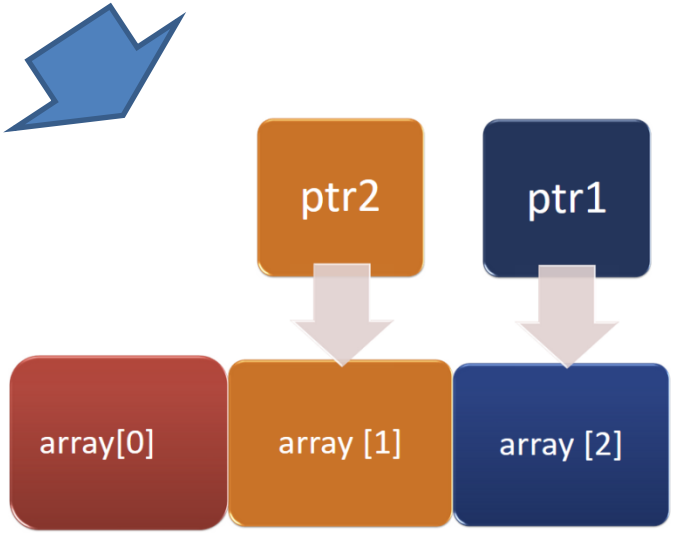
- Try to guess the result of the following operation:

```
ptr1 = ptr1 + 2;
```

Pointers vs. arrays

- Here's the answer





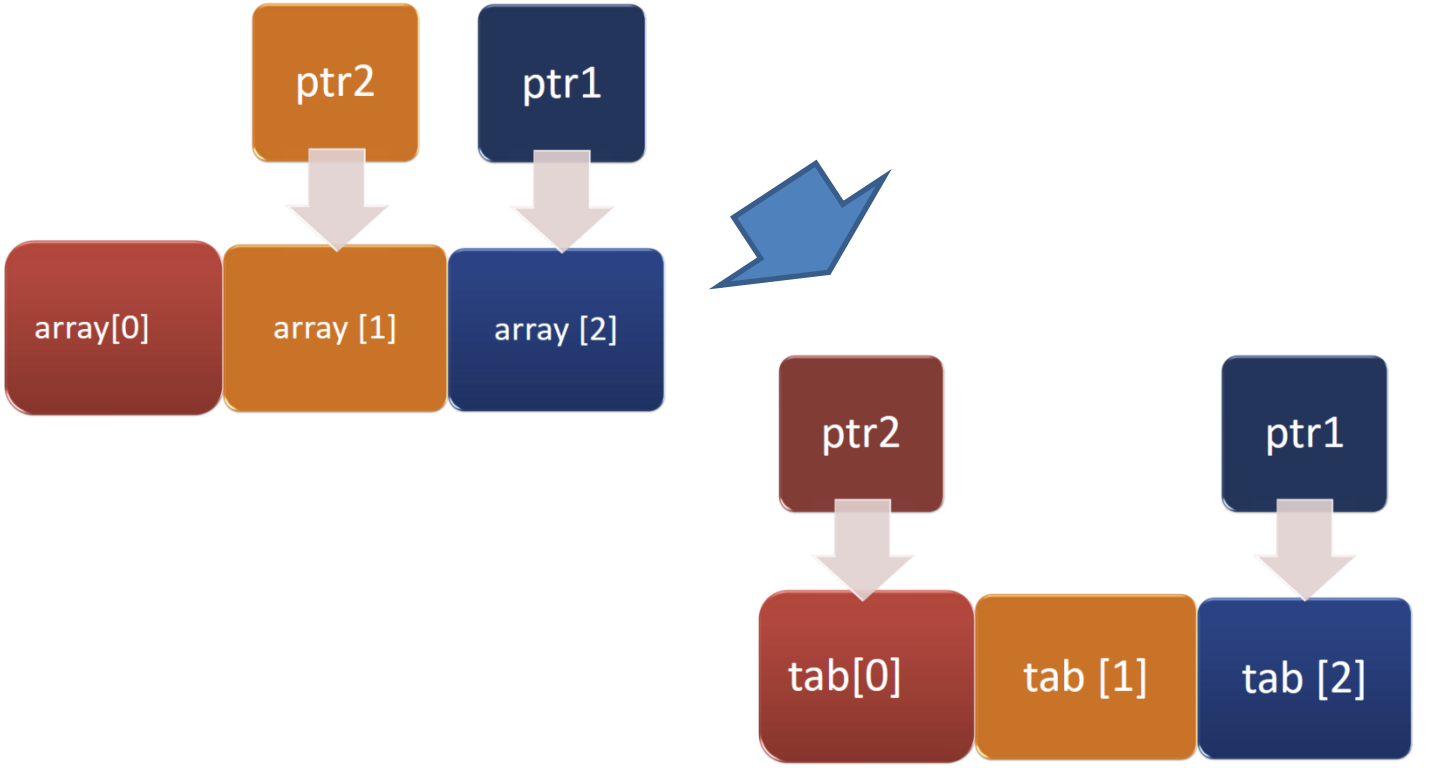
Pointers vs. arrays

- Let's assume that the following operation has been performed. Can you guess the effect?

```
ptr2 = ptr2 – 1;
```

Pointers vs. arrays

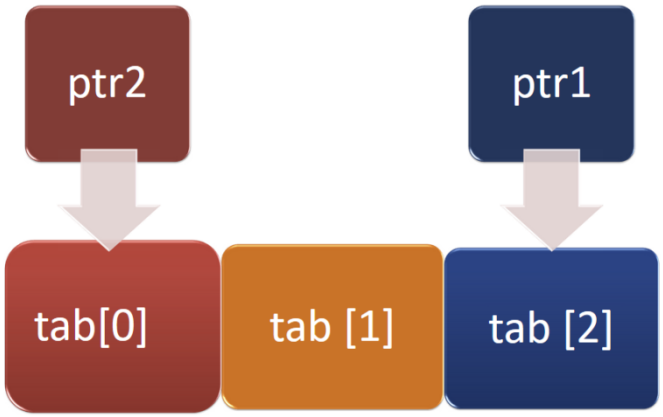
- Let's assume that the following operation has been performed. Can you guess the effect?



Pointers vs. arrays

- Try to determine the result of the following subtraction.

```
ptr1 - ptr2
```



Outline

1. Aggregating data into arrays
    1. Sorting data: in real life and in the computer memory
    2. Initiators - the simple way to set an array
    3. Not only ints
    4. Pointers: another kind of data in the “C” language
    5. Pointers vs. arrays: different forms of the same phenomenon
    6. **The string: a very special vector**
2. Quiz

Arrays of characters – the strings

- **Character arrays** are treated in a special way.
- This array is capable of storing 10 characters.

```
char protagonist[10];
```

- Do you know what we can use similar tables of its kind for?
  - The most obvious example is personal data processing – first names, last names, places of residence, etc. These values are called **strings**.

Arrays of characters – the strings

- We **cannot use the** *sizeof* operator for this purpose. It'll tell us how many characters are occupied by the entire array, but won't tell us how many of them we actually use to store the name.
- This issue is solved in the "C" language in a special way. Every string must end with a special **tag**, something like a flag waving in the wind and announcing: here is the end of the string – all subsequent characters have no meaning.

Arrays of characters – the strings

- According to “C” language conventions, the terminating tag is denoted in the following way (note: it’s a zero, not the letter “O”).
- We call this character an *empty character* or *nul*

'\0'

Arrays of characters – the strings

- How does it work? How do we store the name of our hobbit hero in the array?

```
protagonist[0] = 'B';
protagonist[1] = 'i';
protagonist[2] = 'l';
protagonist[3] = 'b';
protagonist[4] = 'o';
protagonist[5] = '\0';
```

Arrays of characters – the strings

- We can initialize a character array in the same way as any other array, like this:
- Unfortunately, we **can't do this** in regular assignments.

```
char protagonist[10] = { 'B', 'i', 'l', 'b', 'o', '\0' };
```

Arrays of characters – the strings

- There’s another method for initializing character arrays.
- Don't forget – it only works with character arrays.

```
char protagonist[10] = "Bilbo";
```

Arrays of characters – the strings

- Whenever a string appears in the program, the compiler treats it in a very special way and performs the following steps:
  - the compiler **counts how many characters** are inside the string;
  - the compiler **reserves memory** for the string but gets **one character more** than the string's;
  - the compiler copies the entire string from our source code into the reserved memory and appends an empty character at the end;
  - the compiler **treats the string as a pointer** to the reserved memory.

Outline

1. Aggregating data into arrays
    1. Sorting data: in real life and in the computer memory
    2. Initiators - the simple way to set an array
    3. Not only ints
    4. Pointers: another kind of data in the “C” language
    5. Pointers vs. arrays: different forms of the same phenomenon
    6. The string: a very special vector
2. **Quiz**

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i,t[5];
   for(i = 0; i < 5; i++)
      t[i] = 2 * i;
   i = 0;
   for(i = 0; i < 5; i++)
      i += t[i];
   printf("%d",i);
   return 0;
}
```

- ○ the program outputs 14
- ○ the program outputs 12
- ○ the program outputs 13

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i, s = 0, t[5] = {1,2,3,4};
   for(i = 0; i < 5 ; i++)
      s += t[i];
   printf("%d",s);
   return 0;
}
```

- ○ the program outputs 14
- ○ the program outputs 12
- ○ the program outputs 10

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i=10,j=20,*p,s=0;
   p = &i;
   i++;
   (*p)++;
   s = i + j + *p;
   printf("%d",s);
   return 0;
}
```

- the program outputs 54
- the program outputs 64
- the program outputs 44

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i=1,j=2,*p;
   p = &i;
   *p = j;
   *p = 2 * j;
   i = *p;
   printf("%d",i);
   return 0;
}
```

- the program outputs 4
- the program outputs 6
- the program outputs 3

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int t[5] = {1,2,3,4,5};
   int *p1,*p2,s=1;
   p1 = &t[0];
   p2 = &t[4];
   s += *p1 + *p2;
   printf("%d",s);
   return 0;
}
```

- ◯ the program outputs 2
- ◯ the program outputs 5
- ◯ the program outputs 7