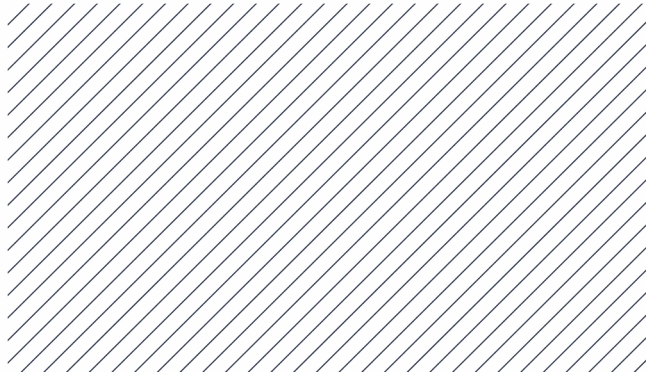
Lecture 6

Pointers

Konstantin Leladze

C++ Basics

DIHT MIPT 2021



Quick recap

I would like to be able to use arrays even if the size is large.

In the case of a regular array, if you use a large size, a RunTime Error (Segmentation Fault) will occur.

```
int a[150]; /// OK
int a[10000000]; /// RE
```

So how to create large arrays?

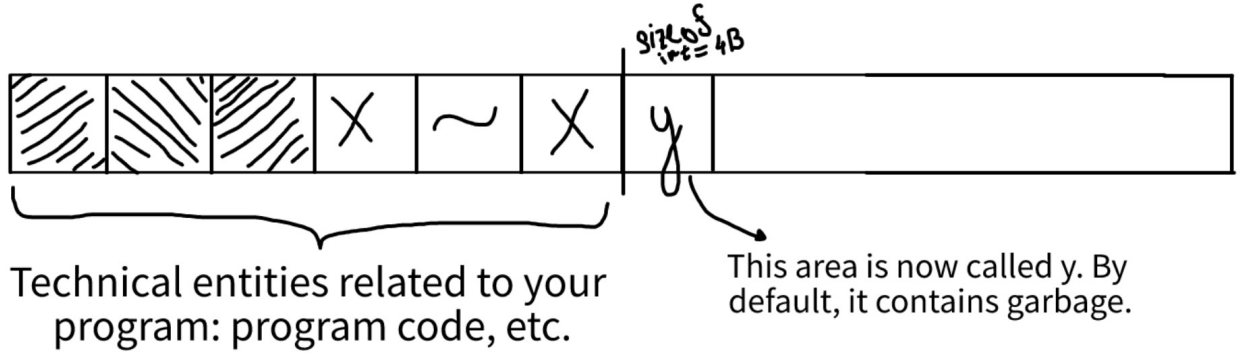
Also:
a [150] = 3;
This code will work, but why ???

Moreover, even the following code works: a[-5] = 4;
Let's figure out what happens at the physical level.

What happens at the physical level

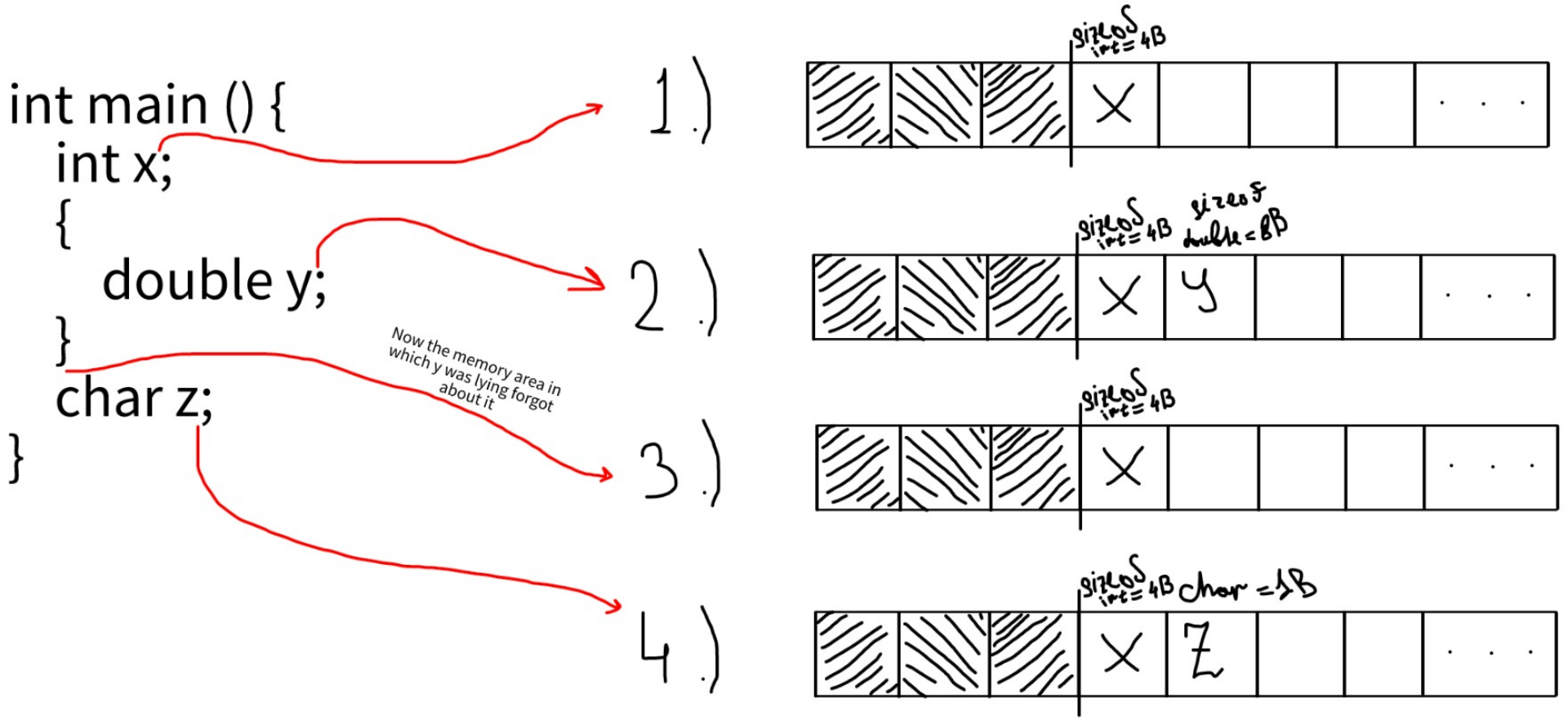
1) When a program is launched, the operating system allocates a fixed amount of memory in the computer's RAM. This is usually 4-8 megabytes.
2) When you declare something in your program, a certain number of bytes is reserved is the special memory area.

```
int main () {
   int y;
}
```



What happens at the physical level

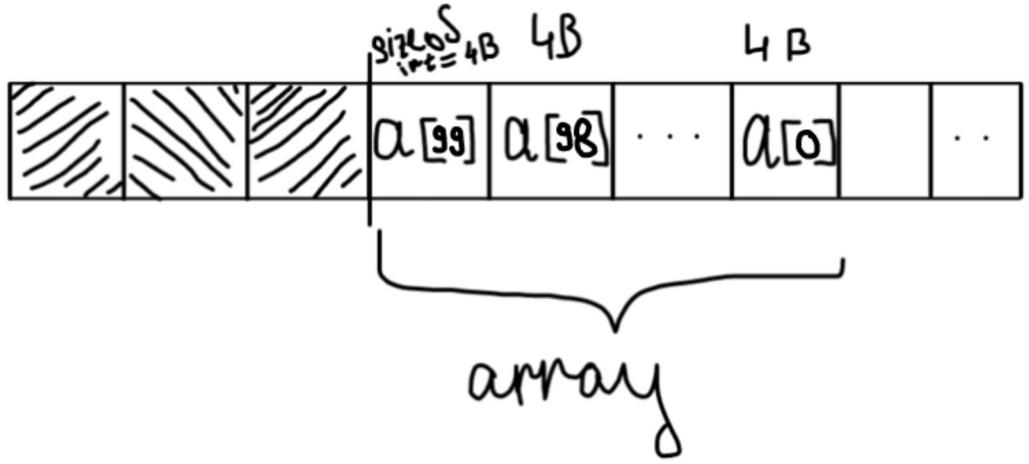
Another example:



What happens at the physical level

Example with an array (note, that array is placed on the stack in the reversed order):

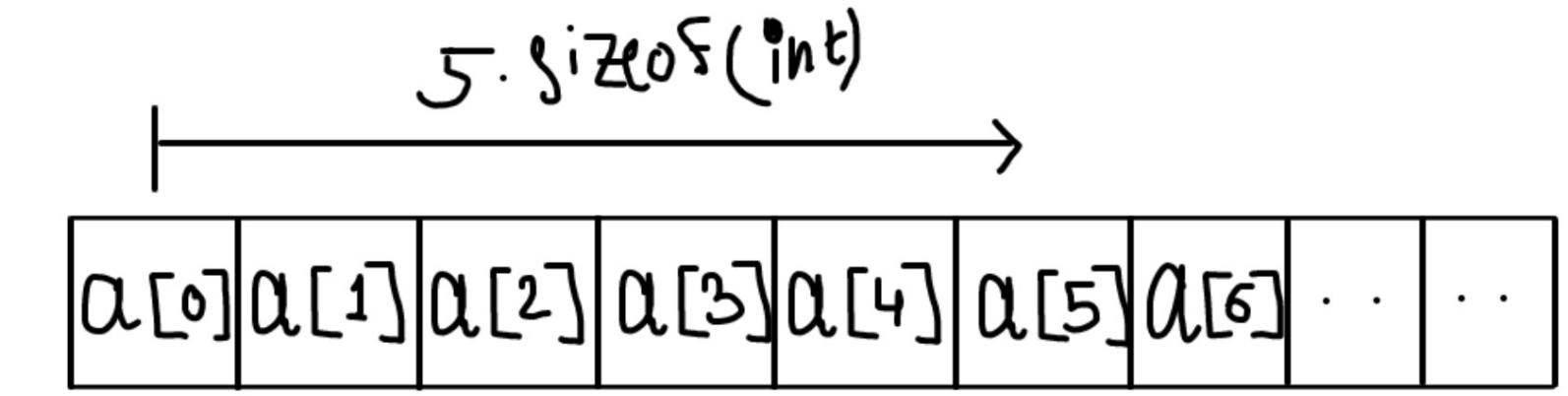
```
int main () {
   int x[100];
}
```



What happens at the physical level

The area of memory discussed is called the stack.

What happens when you access a[5]? The executor understands that he needs to take the sixth element of the array. Now, let's imagine that he has a "coordinate" of the first element of this array a in memory. Then, to get the sixth element of the array, he needs to add 5 to this coordinate, or, in other words, shift it 5 steps to the right. However, you need to take into account that each step must be exactly sizeof(int) = 4Bytes wide.



Pointers

We smoothly arrived at the concept of a pointer. What it is?

Pointer is a special data type that is used to represent different addresses in memory.

A **memory address** is a hexadecimal number representing a coordinate in memory:
`0x7ffea0bbd084`

`int* x;` ← pointer to int, type of x is `int*`

`int*` is a type which stores the memory address in which the int x lies.

Operations with pointers

1) **Unary** operator &, aka **address-of** (don't confuse with the **binary** bitwise operator &).
   Returns the address in memory at which the variable is located.
   precedence: 3
   associativity: right-to-left

2) **Unary** operator *, aka dereferencing
   Returns the value to which the pointer is pointing.
   precedence: 3
   associativity: right-to-left

```cpp
#include <iostream>

int main () {
    int x = 5;
    int* y = &x;

    std::cout <<
        "Value: " << x << std::endl <<
        "Pointer: " << y << std::endl <<
        "Address-of value: " << &x << std::endl <<
        "Dereferenced pointer: " << *y << std::endl;
    return 0;
}
```

```
Value: 5
Pointer: 0x7ffeec0b5ad8
Address-of value: 0x7ffeec0b5ad8
Dereferenced pointer: 5
```

Operations with pointers

3) Incrementing / decrementing pointers.
By adding a number to the pointer, you shift this number of steps to the right (if the number is positive), or to the left (if the number is negative).

```cpp
#include <iostream>

int main () {
    int x = 5;
    int* y = &x;
    std::cout << "Initial address: " << y << std::endl;
    y += 2;
    std::cout << "New address: " << y << std::endl;
    return 0;
}
```

```
Initial address: 0x7ffee00a0ad8
New address: 0x7ffee00a0ae0
```

Operations with pointers

4) Difference of two pointers.
   By subtracting two pointers, you will know the distance between them in memory.
   But this distance will NOT be in the number of bytes, but in the number of elements
   of the pointer type.

So, to find out the number of bytes between two pointers to double, for example, you need to multiply the difference of pointers by sizeof(double).

```cpp
#include <iostream>

int main () {
    double x = 5.;
    double y = 5.;
    double* z = &x;
    double* w = &y;
    std::cout << "Distance in doubles: " << (w - z) << std::endl;
    std::cout << "Distance in bytes: " << (w - z) * sizeof(double) << std::endl;
    return 0;
}
```

```
Distance in doubles: 1
Distance in bytes: 8
```

Operations with pointers

5) Operator [], aka subscript operator.
   Applies to arrays. a[i] returns the array element with index i.
   Precedence: 2
   Associativity: left-to-right

Let's try to print address of an array and its elements.

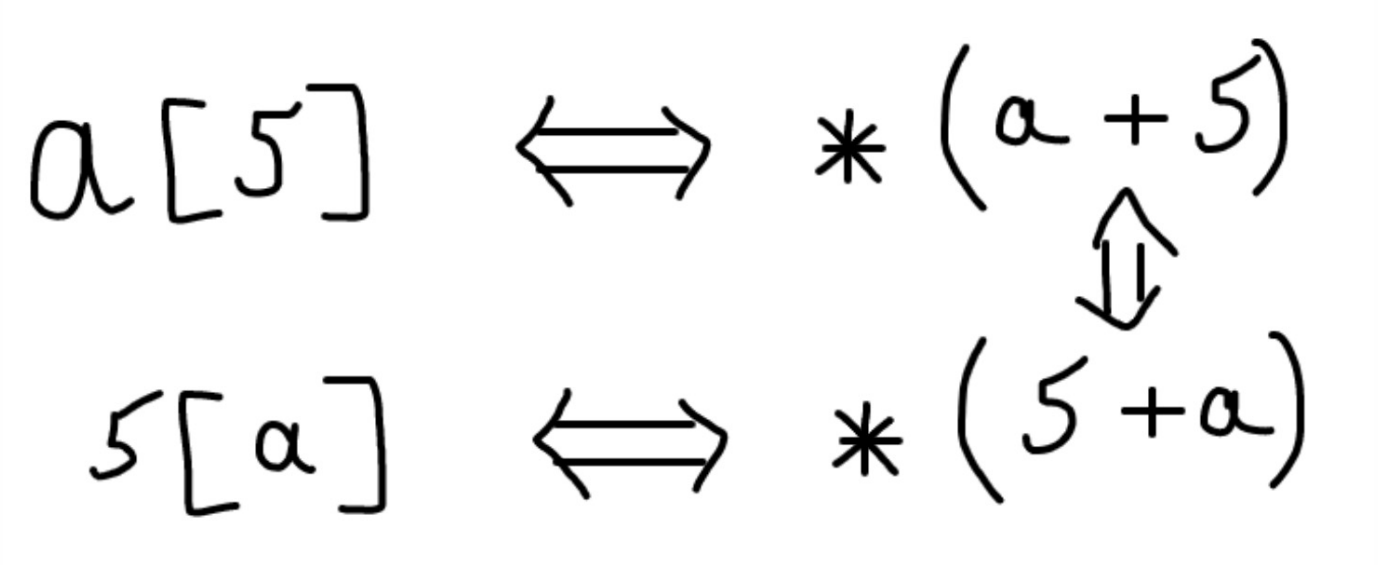
```cpp
#include <iostream>

int main () {
    double a[5];

    std::cout << "Address of an array: " << &a << std::endl;

    for (int i = 0; i < 5; i ++)
        std::cout << "Address of " << i << "-th element: " << &(a[i]) << std::endl;

    return 0;
}
```

```
Address of an array: 0x7ffeedf6bab0
Address of 0-th element: 0x7ffeedf6bab0
Address of 1-th element: 0x7ffeedf6bab8
Address of 2-th element: 0x7ffeedf6bac0
Address of 3-th element: 0x7ffeedf6bac8
Address of 4-th element: 0x7ffeedf6bad0
```

We see an important property: the address of the array coincides with the address of the element with index 0 in it, and the distance between adjacent elements is always 1.

Operations with pointers

In fact, when the [] operator is applied to the array a, the result is calculated as follows:



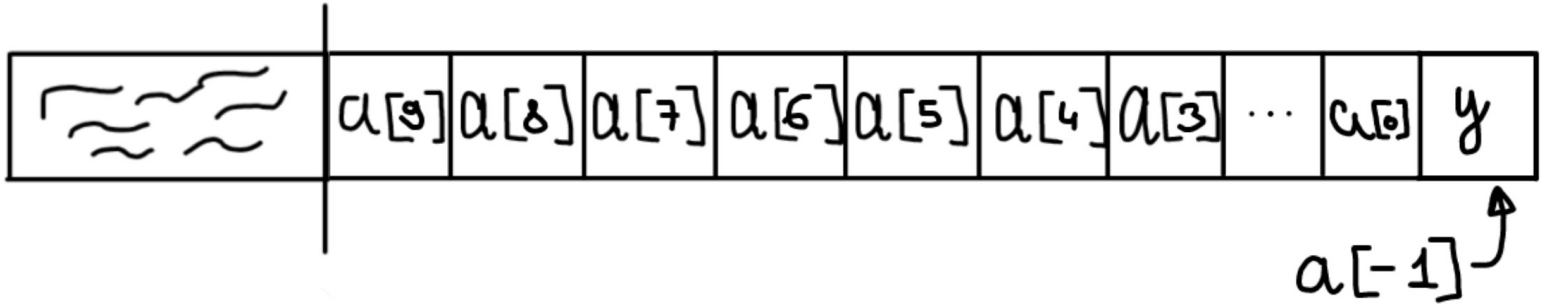
Finally, we understand why the first element in the array has index 0!

Operations with pointers

It is possible to access memory not associated with the current array.

```cpp
#include <iostream>

int main () {
    int a[10];
    int y = 123;
    a[-1] = 5;
    std::cout << "Y value: " << y << std::endl;

    return 0;
}
```

```
Y value: 5
```



Dynamic memory

You cannot allocate very large arrays, because the stack size is limited. But how, then, to create very large arrays ?! For example, a size of 10,000,000 items. The solution is the so-called dynamic memory. The fact is that there is a tool with which you can ask your OS to give you more memory than you currently have available (usually 4-8 megabytes are available).

This is the new and delete operators, which allocate and deallocate heap respectively. Heap memory is located not on the stack, but in another area called the heap.

Operators new and delete

Operators new and delete have two versions:
The first is used to allocate and deallocate one value (cell).
The second is to allocate and deallocate an entire array (sequence) of cells.

```
Operator new:
    Allocates new one cell.s
Operator delete:
    Deallocates one cell.
Operator new[]:
    Allocates new chunk (array) of cells (amount should be provided)
Operator delete[]:
    Deallocates new chunk of cells (amount should be provided)
```

Precedence: 3
Associativity: right-to-left

RULE: All memory allocated by your program must be deallocated!

Operators new and delete, syntax

One cell:

```
int main () {
    int* x = new int(3);
    std::cout << *x << std::endl;
    delete x; /// Don't forget to do!
    return 0;
}
```

3 is a value which will be used to initialize the variable x

Array:

```
int main () {
    int* x = new int[4];
    delete[] x; /// Don't forget to do!
    return 0;
}
```

4 in this case is the size of the array, not the value of it!

Operators new and delete

Now is the time to talk about how to declare arrays that do not end in curly braces.

Example: allocating an array in a function and returning it from a function.

In fact, this can also be done with dynamic memory and using the new and delete operators!

```cpp
#include <iostream>

int* create_array (int n) {
    int* arr = new int[n];

    for (int i = 0; i < n; i ++)
        arr[i] = (i + 1);

    return arr;
}

int main () {
    int n = 0;
    std::cin >> n;
    int* arr = create_array(n);

    for (int i = 0; i < n; i ++)
        std::cout << arr[i] << std::endl;

    delete[] arr; /// Don't forget!
    return 0;
}
```

```
5
1
2
3
4
5
```

Example

```
// Input:
/// 3
/// 1.2 2.3 8.9
/// 3.3 2.5 1.0


// Output:
/// 4.5 4.8 9.9
```

Input two arrays of doubles and output sum of these arrays.

```cpp
#include <iostream>

double* input_array (const int n) {
    double* arr = new double[n];

    for (int i = 0; i < n; i ++)
        std::cin >> arr[i];

    return arr;
}

double* add_arrays (const int n, double* first_array, double* second_array) {
    double* arr = new double[n];

    for (int i = 0; i < n; i ++)
        arr[i] = first_array[i] + second_array[i];

    return arr;
}
```

```cpp
int main () {
    int n = 0;
    std::cin >> n;
    double* first_array = input_array(n);
    double* second_array = input_array(n);
    double* res_array = add_arrays(n, first_array, second_array);

    for (int i = 0; i < n; i ++)
        std::cout << res_array[i] << " ";
    std::cout << std::endl;

    delete[] first_array;
    delete[] second_array;
    delete[] res_array;
    return 0;
}
```

Lecture 6

Pointers

Konstantin Leladze

C++ Basics

DIHT MIPT 2021