HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

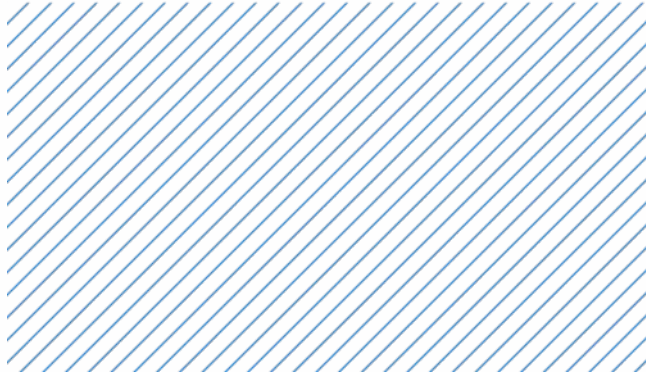
Лекция 2

Основы языка C++, часть 2

Программирование на языке C++

Константин Леладзе

ВШЭ ФКН 2021



Practice

Problem 1:
Input a number and output it’s bits in a sequence.

Practice

Problem 2:
Find the sum of squares of numbers from 1 to N

Input:
4

Output:
30

Explanation:
1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30

Practice

Problem 3:
Find the hypotenuse of a triangle (legs are given)

Input:
3 4

Output:
5

Practice

Problem 4:
Find max and min of two numbers without using conditions

Input:
3 4

Output:
4 3

Practice

Problem 5:
Find the number of paired bits in the binary representation of an integer

Input:
3

Output:
2

Practice

Problem 6:
Let’s consider sets of:

- Digits [10]
- Latin letters (both capital and lowercase ones) [26 + 26 = 52]
- +
- -

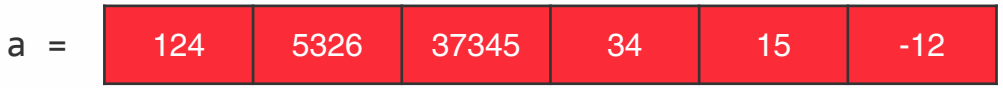
Implement the next functions:

- Input the set
- Output set
- Unite two sets
- Intersect two sets
- Invert set
- Calculate symmetric difference between two sets
- Calculate difference between two sets

Arrays

Variable is like a cell in a storage room

But what if we want to take an arbitrary amount of these cells?



An array is special type of a variable. It contains several variables of the same type at once.

Moreover, the elements of the array are ordered - each has its own number.

Only the entire array has a name, the elements have only an ordinal number in this array

Arrays

Items are numbered starting at zero. The number of an element in an array is called its index

a =

0	1	2	3	4	5
124	5326	37345	34	15	-12

Array size is 6!

You can think of the index as the distance from the leftmost element. On the ruler, lines are also numbered from zero.

An array element is accessed through square brackets: a [i]

```
std::cout << a[1] << std::endl;
a[2] = 0;
a[3] = a[2] + a[1];
std::cout << a[3] << std::endl;
```

Arrays

When creating an array, you need to specify its size:

```
int a[6];
```

By default, as in the case of an uninitialized variable, all array elements contain so-called garbage. That is, each element has no specific meaning. Therefore, the arrays need to be initialized:

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;
a[5] = 6;
```

Array and for loop are best friends =)

Let's try to *initialize* the array with numbers from 1 to 10:

```
int a[10];
for (int i = 0; i < 10; i ++)
    a[i] = i + 1
```

Practice

Task 1: A sequence of n numbers is entered on the keyboard. Print numbers in reverse order.
Input: First, the number n itself, then a sequence of n numbers
Output: The same n numbers in reverse order

Task 2: A sequence of n numbers is entered on the keyboard. Determine if it is a palindrome.
Input: First, the number n itself, then a sequence of n numbers
Output: Whether the string is a palindrome or not

Task 3: A number n is given, which is the size of a square matrix. It is necessary to assign to each diagonal its distance from the main one.
Input: the number n
Output: the required matrix

Arrays and const qualifiers

If we combine array modifier and const qualifier, we'll create a const-qualified array.

```
const int a[6];
```

But in this case, we'll get a compilation error, because a is uninitialized.

In order to initialize it, we should use initializer lists:

```
const int a[6]{1, 2, 3, 4, 5, 6};
```

or:

```
const int a[6] = {1, 2, 3, 4, 5, 6};
```

Declaration point of an array

```
int x[x];
```

Compilation error, x is undefined.

```
int x = 2;
{
    int x[x];
}
```

Ok, array size is 2

Arrays

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?

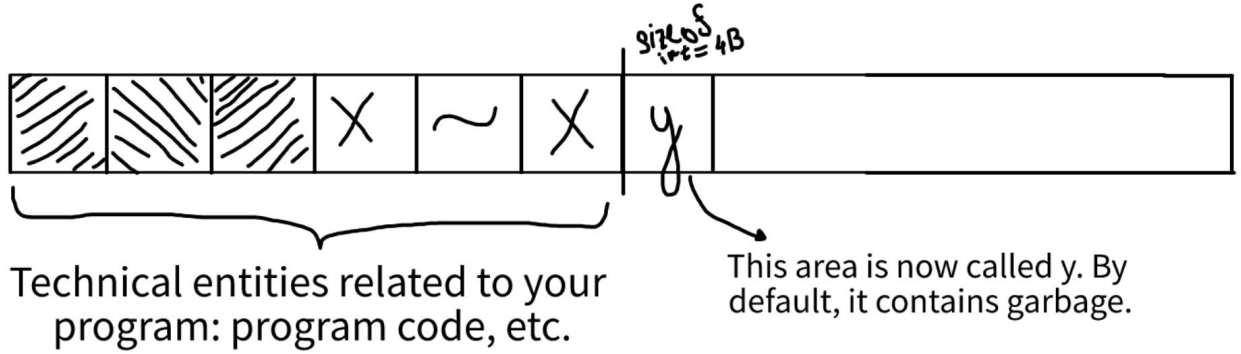
Also:
a [150] = 3;
This code will work, but why ???

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

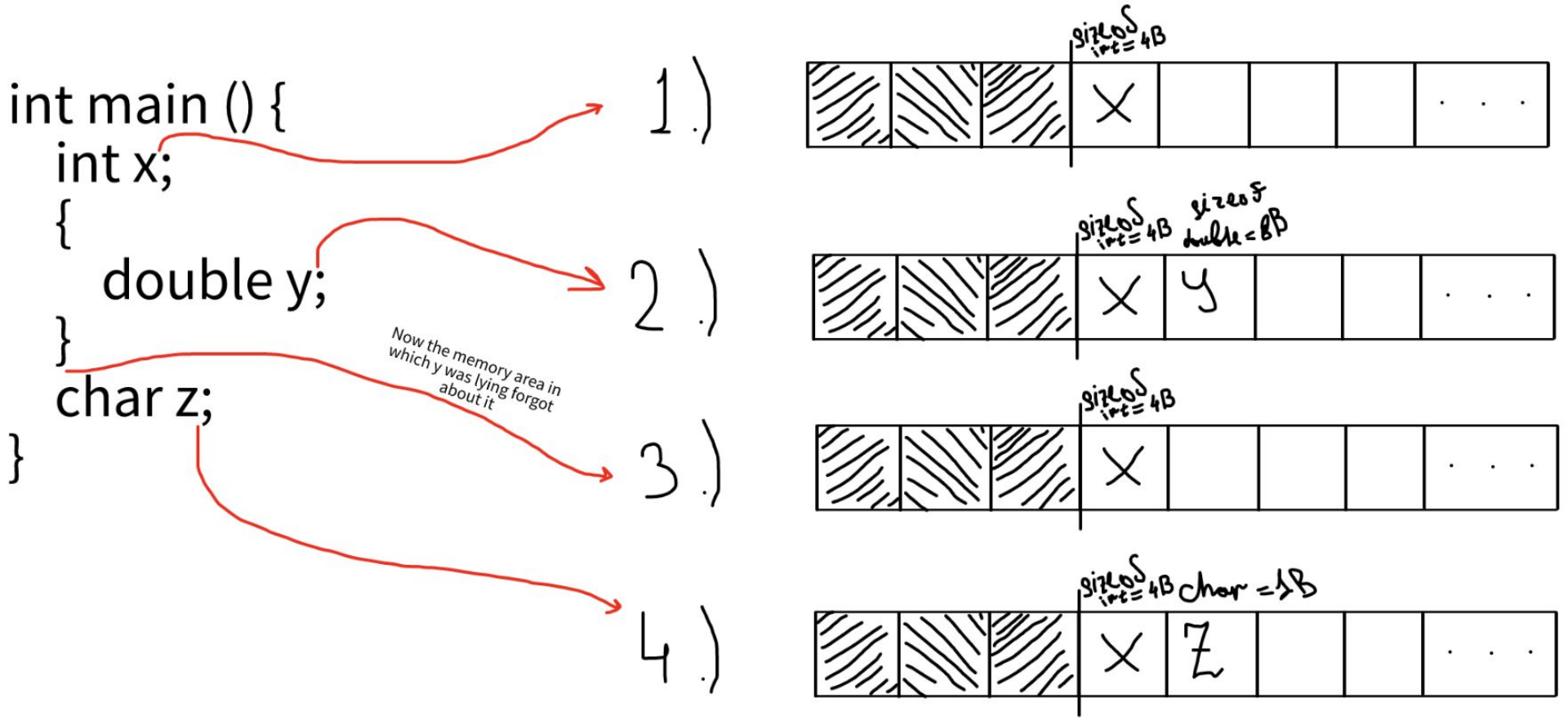
1) When a program is launched, the operating system allocates a fixed amount of memory in the computer's RAM. This is usually 4-8 megabytes.
2) When you declare something in your program, a certain number of bytes is reserved is the special memory area.

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



What happens at the physical level

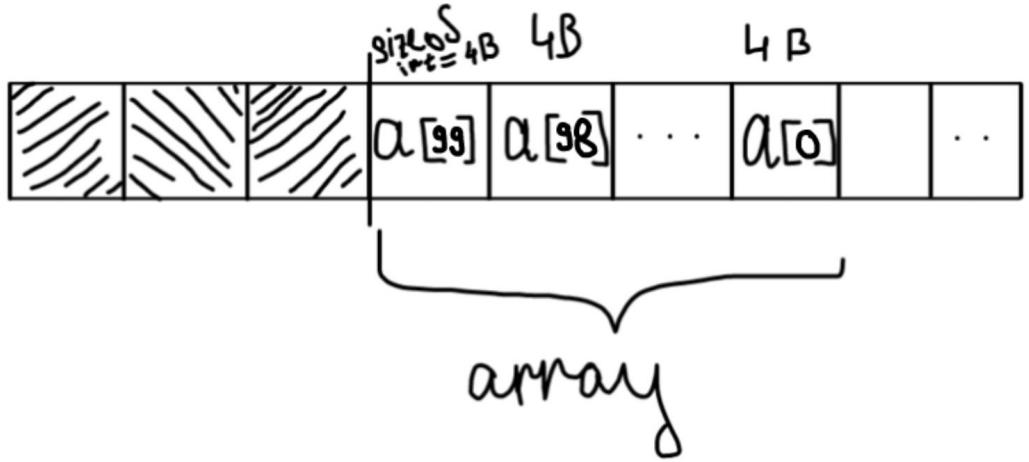
Another example:



What happens at the physical level

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

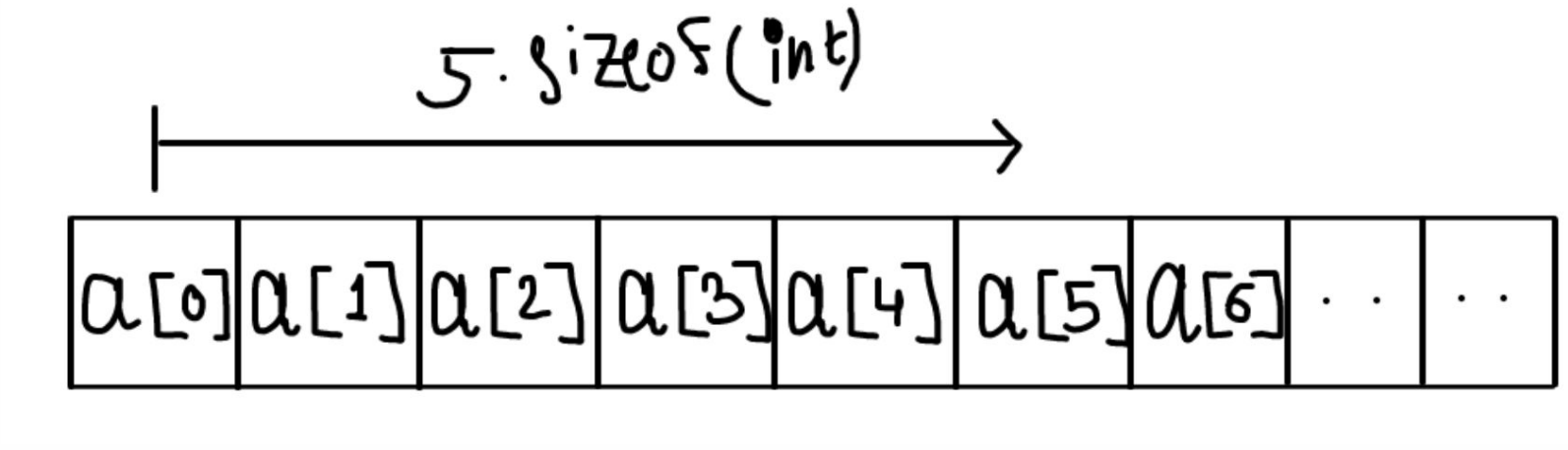
```
int main () {
   int x[100];
}
```



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.

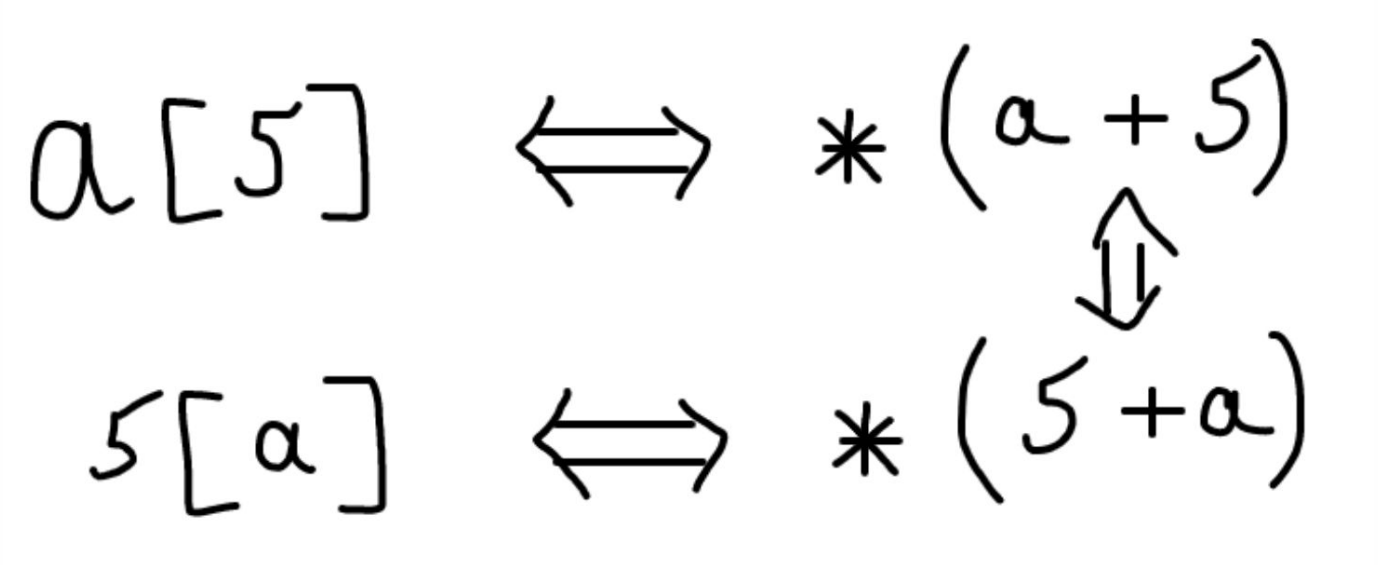
```cpp
#include <iostream>

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:



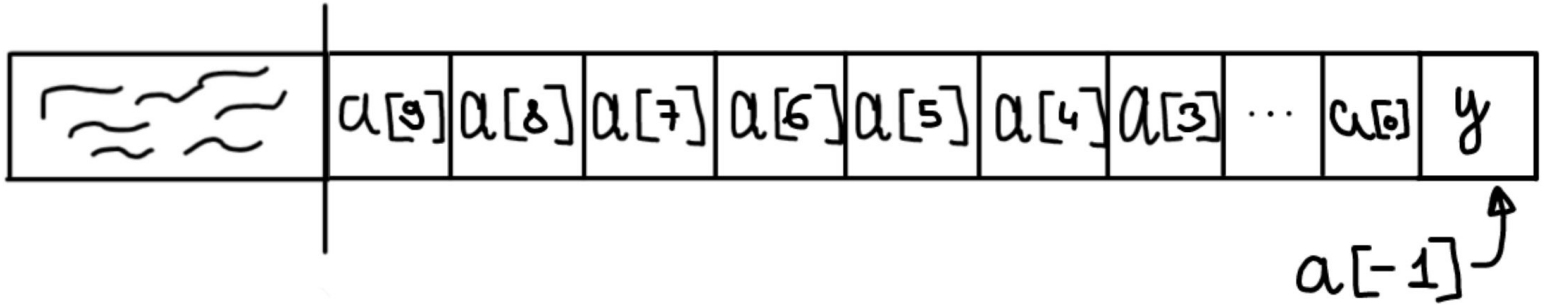
Finally, we understand why the first element in the array has index 0!

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;
}
```

Const qualifier and pointers

Let me remind you that C ++ has a const keyword, which is a variable qualifier. If we create a variable and add the const qualifier to it, that variable cannot be modified.

```
int main () {
    const int x = 3;
    x = 5; /// CE
    return 0;
}
```

The const qualifier can be applied to pointers too! However, in this case there is a question: what exactly will we prohibit modifying if we add the word const: the pointer itself (address), or the value it points to ?! In fact, pointers are of four types:

- A regular pointer. You can modify both its value and the value to which it points.
- Constant pointer. Modifying its value is not possible, but it is possible to modify the value to which it points.
- Constant pointer. Modifying its value is possible, but it is not possible to modify the value to which it points
- A constant pointer to a constant. Modifying neither the value of the pointer nor the value to which it points is allowed.

```
int main () {
    int x;
    const int y = 3;

    int* ptr = &x;
    int* const const_ptr = &x;
    const int* ptr_const = &y;
    const int* const const_ptr_const = &y;

    return 0;
}
```

Const qualifier and pointers

A pointer to a const returns a const when dereferenced.

A pointer to a const cannot be assigned to an normal pointer, but vice versa is allowed.

```
int main () {
    int x;

    const int* const_ptr = &x;
    int* ptr = &x;

    ptr = const_ptr;
    const_ptr = ptr;

    return 0;
}
```

Conclusion: any operation should not underpromote (*get rid of*) the **const** qualifier.

Swap

Let's imagine the following problem: we want to swap the values of variables.
Moreover, we want to create exactly the function that will do this.

Formally, we want to create a function that will take two variables and swap their values.
Let's try to implement such a function.

```
void swap (int x, int y) {
    int t = x;
    x = y;
    y = t;
}

int main () {
    int x = 2;
    int y = 3;
    swap(x, y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

But it doesn’t work =(
Why?

```
2 3
```

Swap

When we pass an argument to a function by value (as in the previous example), in fact, we are creating a copy of the variable, and not passing this particular variable! So, in previous implementation of the swap function, we do not work with the initial variables x and y, but with their **copies**.
But how, then, can we access the original variables x and y!? Well, for example, we can pass the addresses of these variables to the function, and not just copy their values! To do this, let's change the signature of the swap function to receive pointers to the original two variables, not copies of them!

```
void swap (int* x, int* y) {
    int t = *x;
    *x = *y;
    *y = t;
}

int main () {
    int x = 2;
    int y = 3;
    swap(&x, &y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

And this time everything works as it should!

```
3 2
```

Swap

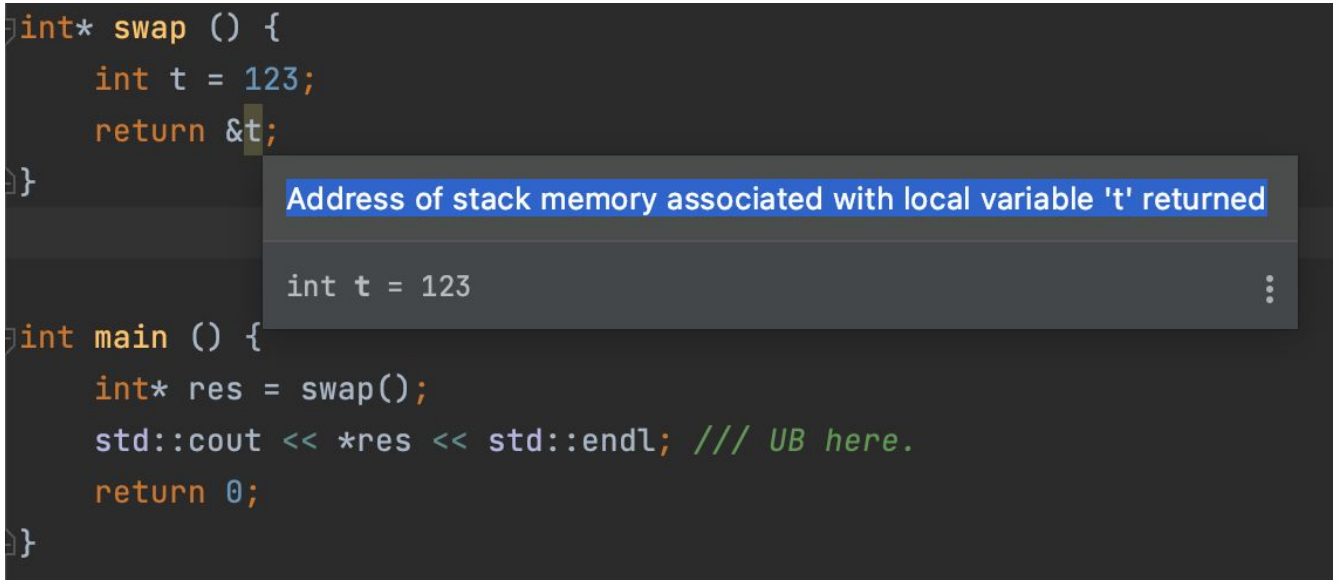
We can improve the code a bit by keeping the **const** rule, which states: everything that can be const must be const.

For example, in this problem, we can make **const** pointers (but NOT pointers to **const**).

```
void swap (int* const x, int* const y) {
    int t = *x;
    *x = *y;
    *y = t;
}
```

References

In fact, before the invention of C ++, the method I showed was the only method for passing initial variables to functions. This method is not very simple, as it requires a lot of dereferencing and address-of operations. In addition, you always need to make sure everything is fine with the pointer. For example, if you accidentally delete an object and then refer to it by the pointer, it'll be an UB:



But with the invention of C ++, everything changed and the concept of references emerged.

References

In Python, for example, if we create a list and create a variable to which we assign the value of this list, we will NOT copy the list, but create a reference to it!

```python
a = [1, 2, 3]
b = a
print(len(a), len(b))

a += [4]
print(len(a), len(b))
```

```
3 3
4 4
```

You will find similar behavior in some cases in Java, JavaScript, TypeScript, and many other programming languages. In them - the concept of a link is built into the language, but you have a way to control whether to create a reference to an object, or make a copy of it.

Similarly in C ++, we can create references to objects (variables). However, in C ++, by default, the copy is getting created, not creating a reference. This is what distinguishes C ++ from Python and the other languages list above.

```cpp
int main () {
    int a = 2;
    int b = a;
    a += 124;
    /// b is a copy here, not a reference, so
    /// its value hasn't been changed
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

```
126 2
```

References

```
int main () {
    int a = 3;
    int &b = a;
    return 0;
}
```

To create a reference to a variable in C ++, you must use the ampersand character in declaration (not to be confused with the address-of operator!)

& here denotes a modifier of type **int** (as in the case of a pointer). That is, the type of variable **b** is int **&** (not **int**!).

Now, any action with variable b will also change variable a!

```
int main () {
    int a = 3;
    int &b = a;
    b += 123;
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

```
126 126
```

References

```
void swap (int& x, int &y) {
    const int t = x;
    x = y;
    y = t;
}

int main () {
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```

References can be passed to functions. In this case, we will NOT create a copy, but pass the variable itself to the function. This way we can change the value of the original variable without using pointers!

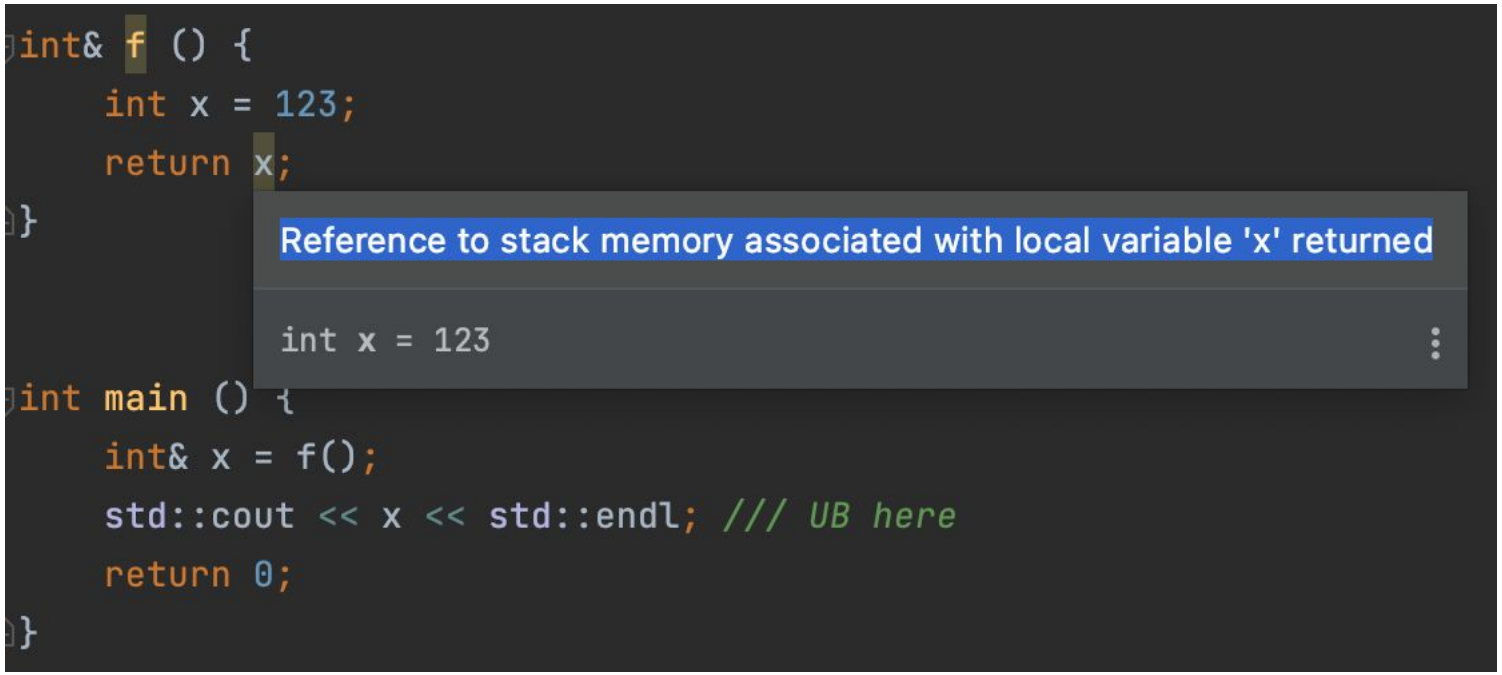
In this case, everything is fine

References

Moreover, references can not only be received by functions, but also returned from them!

However, this trick will be useless to us before we look at classes.

Moreover, there is one very serious mistake associated with this that many programmers make: returning a reference to a local object from a function



Const qualifier and references

Just like with pointers, we can create references to constants.
In this case, the rule from the first slide is also fulfilled.

```
int main () {
    const int& x = 3;
    int y = x;
    int &z = x;
    const int &t = y;
    const int tt = y;
    return 0;
}
```

Another rule: const and reference must always be initialized

```
int main () {
    int& x;
    const int y;
    return 0;
}
```

Another example:

```
int main () {
    const int *p = new int[10]; /// OK
    const int *q = new const int[10]; /// CE
    return 0;
}
```

Casts

There is a so-called C-style cast operator, which allows you to convert variables of different types. But how **exactly** does this transformation work?

In fact, the C-style cast operator isn't quite often used nowadays, because its behavior is not specific enough.

Instead, in new versions of C ++, 4 new operators are used, of which today we will discuss only 3.

```
int main () {
    int a = 123;
    double x = (double)a / 2.;
    std::cout << x << std::endl;
    float z = (float)x + 0.4f;
    std::cout << z << std::endl;

    return 0;
}
```

Casts

- static_cast: This is a compile-time conversion. If static_cast fails to convert the original variable to the correct type, we will get a compilation error.

- reinterpret cast - byte-level conversion. C ++ will simply stop treating the original object as the type it was originally, and will hang a new type on it. This is the lowest-level conversion you can do in C ++. You should be careful with it, and use it only in the most special cases. In the next lesson, I'll show you an example.

```cpp
int main () {
    int a = 123;
    double x = static_cast<double>(a) / 2.;
    std::cout << x << std::endl;
    float z = static_cast<float>(x) + 0.4f;
    std::cout << z << std::endl;

    return 0;
}
```

```cpp
int main () {
    double x = 312.52;
    unsigned long long z = *reinterpret_cast<unsigned long long*>(&x);
    std::cout << x << " " << z << std::endl;

    return 0;
}
```

```
312.52 4644205526174211768
```

Casts

- const_cast is a conversion "through" a constant. This is the only cast that violates the underpromotion rule. You have to be careful with const_cast because it can lead to an error.
- dynamic_cast is a run-time conversion that is applied to polymorphic (virtual) types. We'll be talking about it in the next semester.

```
int main () {
    int x = 312;
    const int& y = x;
    int& z = const_cast<int&>(y);
    return 0;
}
```

```
int main () {
    const int x = 312;
    const int& y = x;
    int& z = const_cast<int&>(y); /// UB
    return 0;
}
```

Problem

A sequence of 2D points is specified as a sequence of pairs **(x, y)**.

It is necessary to find the perimeter of the polygon formed by a given set of points.

Points are given in order of counterclockwise traversal relative to the center of the polygon.

```cpp
#include <iostream>
#include <cmath>

double dot_product (const double x1, const double y1, const double x2, const double y2) {
    return x1 * x2 + y1 * y2;
}

int main () {
    size_t n;
    std::cin >> n;
    double* const x = new double[n];
    double* const y = new double[n];
    double perimeter = 0.;

    for (size_t i = 0; i < n; i ++)
        std::cin >> x[i] >> y[i];

    if (n >= 3)
        for (size_t i = 0; i < n; i ++) {
            const size_t j = (i + 1) % n;
            const double v_x = x[j] - x[i];
            const double v_y = y[j] - y[i];
            perimeter += std::sqrt(dot_product(v_x, v_y, v_x, v_y));
        }

    std::cout << perimeter << std::endl;

    delete[] x;
    delete[] y;
    return 0;
}
```

HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 2

Основы языка C++, часть 2

Программирование на языке C++

Константин Леладзе

ВШЭ ФКН 2021