Datatypes and Objects. Primitive, composite and user datatypes. Scope of variables (objects). Constants. Basic input/output. Introduction to streams. Control flow statements: conditional (if, switch); loops (while, do..while, for).

Standard C++ Data Types

An array is a finite named sequence of values of the same type. The array elements in memory are arranged linearly and continuously.

```
type arr_name[N];
```

type – type of array elements;

N – number of elements. N must be determined at compile time.

Standard C++ Data Types

You can access array elements using indexes:

```
type arr[N];
type val = arr[5];
arr[0] = val;
```

The numbering of array elements starts from zero.

Standard C++ Data Types

The array can be initialized as follows:

type arr[N] = {a_0, a_1, …, a_M};

where $M \leq N - 1$.

If $M < N$, then the remaining elements are initialized to zero values.

Standard C++ Data Types

```
int arr1[5] = {1, 2, 3, 4, 5};   1 2 3 4 5


int arr2[5] = {1, 2};                            1 2 0 0 0


int arr3[5] = {0};                                      0 0 0 0 0
```

Standard C++ Data Types

Arrays can be multidimensional:

```
type arr_name[N][M]…[K];
```

In this case, the values of all dimensions must be determined at the compilation stage.

Standard C++ Data Types

Multidimensional arrays can be initialized with subarrays. In this case, the elements included in a certain subarray are grouped using curly braces:

```
type arr1[N][M] = { {1, 2, 3}, {4, 5, 6, 7} };
```

In this case, the remaining elements will also be initialized to zero values.

Standard C++ Data Types

A multidimensional array can also be initialized similarly to a one-dimensional one. In this case, the array elements are initialized sequentially.

```
type arr1[N][M] = { 1, 2, 3, 4, 5, 6, 7 };
```

Standard C++ Data Types

```
int arr1[3][2] = { {1}, {2, 3}, {4, 5} };
```

```
1 0
2 3
4 5
```

```
int arr2[3][2] = { 1, 2, 3, 4, 5 };
```

```
1 2
3 4
5 0
```

Input Output C-style

Input functions:

scanf

getchar

gets

Output functions:

printf

putchar

puts

Input Output C-style

To use C-style I/O functions, you must include a header file stdio.h

```
#include <stdio.h>
```

Input Output C-style

- gets – reads characters from standard input (stdin) and stores them as a C string (char array)

- getchar – reads the next character from standard input

- scanf – reads data from standard input and saves according to the specified format

Input Output C-style

```
int scanf(const char* format, …)
```

format – a string specifying the type of data to be read.

Variables for storing values are passed as a pointer.

Input Output

Basic format specifiers

format specifier	**meaning**	**format specifier**	**meaning**
%c	Reading a character	%p	Pointer
%s	Reading a line	%o	Number in octal format
%ws	Reading a string in extended encoding	%x	Number in hexadecimal format
%d	Signed decimal number	%u	Unsigned integer
%f	Floating point number		

Input Output C-style

```
float a;
scanf(“%f”, &a);

int b, c;
int* ptr = &c;
scanf(“%d %d”, &b, ptr);

int d = getchar();
```

Input Output C-style

- puts - Writes a C-style string to stdout and adds a newline at the end.
- putchar – writes a character to standard output (stdout).
- printf – writes a string to standard output (stdout) in the specified format.

Input Output C-style

```
int printf(const char* format, …)
```

format – a string written to standard output (stdout).

This line may contain format specifiers similar to scanf, which will be replaced by the argument values when printed to the screen.

Input Output C-style

```
float a = 1.23;
printf(“a = %f\n”, a);          a = 1.230000

int b = 122, c = 133, *ptr = &c;
printf(“b = %d\nc = %d\nptr = 0x%p”, b, c, ptr);
```

```
b = 122
c = 133
ptr = 0x11223344
```

Input Output

To use C++ style I/O functions, you need to include a header file iostream

```
#include <iostream>
```

Input Output

To use C++-style input, you use the standard input stream object std::cin and the stream read operator >>.

To use C++ style output, you use the standard output stream object std::cout and the stream write operator <<.

When connecting the std namespace (using namespace std;), you don't have to write “**std::**” .

Input Output

```
float a;
std::cin >> a;

int b, c;
std::cin >> b >> c;
```

Input Output

When using the standard output object std::cout, it is convenient to use manipulators to print numbers in various formats, terminate a line, etc.

Manipulator	Meaning
`std::endl`	Line termination
`std::dec`	Displaying a number in decimal format
`std::oct`	Printing a number in octal format
`std::hex`	Printing a number in hexadecimal format

Input Output

```
float a = 12.345, d = 1.23;
std::cout << “a = ” << a << std::endl;
```

```
a = 12.345
```

```
int b = 123, c = 345;
std::cout << “b = ” << b << “\n” << “c = 0x ” << std::hex << c << std::endl << “d = ” <<
std::dec << d;
```

```
b = 123
c = 0x159
d = 1.23
```

Examples

```
double c = 0.1 + 0.2;


//C++ style
#include <iomanip>
...
std::cout<<std::setprecision(17)<< c << std::endl;
```

Результат: 0.30000000000000004

C++ language operators

<u>Operator classification:</u>

Expression operators (discussed earlier);

Branch operators;

Loop operators;

Control transfer operators.

Branch Operators

Conditional if statement

Used to branch the calculation process into two directions.

```
if (logic_expression_1)
        operator_1;
[else if (logic_expression_2) operator_2;]
[else operator_3;]
```

Branch Operators

```
int a = 5;

if (a == 3)
        std::cout << “a == 3” << std::endl;
else if (a == 5){
        std::cout << “a == 5” << std::endl;
        a++;
}
else
         std::cout << “unknown value” << std::endl;
```

Branch Operators

Multiple choice switch operator

Branches the calculation process into several directions.

```
switch (expression){
        case const_expression_1: [operators_1,...]
        …
        case const_expression_N: [operators_N, ...]
        [default: operators...]
}
```

Операторы ветвления

expression must be an integer.

The switch statement evaluates expression and transfers control to the case branch whose expression matches the value of expression.

Branch Operators

The optional default branch is executed if expression does not match any const_expression.

If the exit from any branch is not explicitly specified, then all other branches are executed sequentially.

Exiting a branch can be done with a break or return statement.

Branch Operators

```
int a = 1;
…
switch(a){
        case 1:
                std::cout << “First branch” << std::endl;
                a++;
        case 3:
                std::cout << “Second branch” << std::endl;
                break;
        default:
                std::cout << “Unknown value” << std::endl;
}
```

Loop Operators

A cycle is a control structure designed to organize repeated execution of a set of instructions.

Iteration is one pass of the loop.

The loop ends if its continuation condition is not met.

Loop Operators

A counter loop is a loop that runs a specified number of times.

```
for(initialization; expression; change)
        operator;
```

A loop counter variable can be declared directly in the initialization section of the loop. The scope of such a variable is the body of the loop.

Loop Operators

```
int arr[10];

for(int i = 0; i < 10; i++)
{
        arr[i] = i;
        std::cout  << arr[i] << “ ”;
}
```

Loop Operators

```
int arr[10];

for(int i = 9; i >= 0; i--)
{
        arr[i] = i;
        std::cout  << arr[i] << “ ”;
}
```

Loop Operators

In a counter loop, several variables of the same type can be declared:

```
for(int i = 0, j = 5; (i<=5) & (j>0); i++,j--)
{
        …
}
```

Loop Operators

Loops can be nested:

```
for(int i = 0; i < 3; i++)
        for(int j = i; j < 3; j++)
        {
                std::cout<< i <<“ ”<< j << std::endl;
        }
```

```
result:
    0 0
    0 1
    0 2
    1 1
    1 2
    2 2
```

Loop Operators

A loop with a precondition is a loop that runs until the expression condition is false.

```
while(expression)
{
        …
}
```

Loop Operators

```
double eps = 1.0;

while(1 + eps > 1)
{
        eps /= 2.0;
}
```

Loop Operators

A loop with a postcondition is a loop in which the expression condition is checked after the loop body has been executed. This loop is always executed at least once.

```
do
{
        …
} while (expression);
```

Loop Operators

```
int odd_num;

do{
        std::cout<<“Enter odd number:”<<std::endl;
        std::cin >> odd_num;
} while ((odd_num % 2) != 1);
```

Loop Operators

Infinite loops:

```
while(1) { ... }

while(true) { ... }

for(;;) { ... }
```

Control Transfer Operators

break statement

Used inside branch and loop statements to provide a jump to a point in the program immediately following the statement that contains the break.

Control Transfer Operators

```
for(int i = 0; i < 10; i++)
{
        if(i == 5)
                break;
        std::cout << i << std::endl;
}
```

```
result:
    0
    1
    2
    3
    4
```

Control Transfer Operators

```
for (int i = 0; i < 5; i++){
        for (int j = 0; j < 5; j++){
                if (j == 3)
                        break;
                std::cout << j << “ ”;
        }
        std::cout << std::endl;
}
```

```
result:
        0 1 2
        0 1 2
        0 1 2
        0 1 2
        0 1 2
```

Control Transfer Operators

continue

The operator skips all instructions remaining until the end of the loop and transfers control to the beginning of the next iteration.

Control Transfer Operators

```
for(int i = 0; i < 10; i++)
{
        if(i == 5)
                continue;
        std::cout << i << std::endl;
}
```

```
result:
    0
    1
    2
    3
    4
    6
    7
    8
    9
```

Control Transfer Operators

return

Completes the execution of a function and transfers control to the point of its call.

An operator can also return a value by passing it to the calling function.

```
return [value];
```

Control Transfer Operators

```
for (int i = 0; i < 5; i++){
        for (int j = 0; j < 5; j++){
                if (j == 3)
                        return;
                std::cout << j << “ ”;
        }
        std::cout << std::endl;
}
```

```
result:
     0 1 2
```

Control Transfer Operators

goto

An unconditional jump operator to a specific point in the program, indicated by a label Label.

```
goto Label;
...
Label: operators;
```

Control Transfer Operators

```
for (int i = 0; i < 5; i++){
        if (i == 3) goto Label;
        std::cout << i << “ ”;
}
Label:
std::cout << “after label” << std::endl;
```

```
result:
    0 1 2 after label
```

Control Transfer Operators

The scope of a label is the function in which it is declared.

Using goto, you cannot jump into the body of other functions, inside loops, or jump over blocks of code containing declarations with initialization.

Control Transfer Operators

- Using the goto statement:
  - Exit from a large number of nested loops and switch statements;
  - Transition from several places in a function to one.

Control Transfer Operators

```
int res = InitPCI();
if (res != SUCCESS)
        goto ErrorHandler;

...
res = InitUSB();
if (res != SUCCESS)
        goto ErrorHandler;
...
ErrorHandler:
        FreeResources();
        CloseHandlers();
```

Control Transfer Operators

The use of the goto statement should be avoided whenever possible.