POINTERS.

Pointer is the variable that stores the address of the data, and not the value. Pointer declaration: **<type> * <variable name>**

In C/C++ pointers are used for the following purposes:

1. Mapping data structures in memory.
2. Dynamic memory allocation and management.
3. Access to individual elements of composite data structures (for example, arrays).

Two basic operations used for working with pointers:

* – reference to the value of the data according to the pointer address,

& – getting the address where the value is stored.

Example:

```
#include <stdio.h>
main()
{
int Z,*Y;
Y = &Z;
Z = 100;
printf ("Directly accessed Z value: %d\n", Z);
printf ("Z value accessed by pointer: %d\n",*Y);
printf (" Address of Z variable: %p\n",&Z);
printf ("Address of Z variable received by using a pointer: %p\n", Y);
}
```

Rezults:

```
Directly accessed Z value: 100
Z value accessed by pointer: 100
Address of Z variable:  85B3:0FDC
Address of Z variable received by using a pointer:  85B3:0FDC
```

Pointer initialization: **int kint1, rod1 = (int*) 2002, *rod2 = &kint1;**
The pointer itself also has an address, so this operation can be also done:

rod1 = (int*)&rod2;

Restrictions for working with pointers:

1. The constants have no memory addresses, so the constant‘s address can not be set. Improper operations:

****123 = kint1;***
kint1 = &123

2. The address of the arithmetic expression can not be determined, for example, this is illegal:

   rod1 = &(kint1 +10);

3. The address of the registry type variable can not be determined:

```
unsigned register kint2;
unsigned int *rod3;
rod3 = &kint2;
```

Other operations with pointers (address arithmetic):

=, ++, - -, +, - , ==, !=, <, >, <=, >=

When arithmetic operations with pointers are performed, their result depends on the type of the pointer, for example:

int n;

rod = rod + n;

then the result is calculated as:

rod = rod + n * sizeof(int);

<u>Example</u>:

```
    #include <stdio.h>
    void main(void)
    {
    int near * rod1 = (int*) 100;
    int near * rod2 = (int*) 200;
    rod1++;
    rod2 - = 10;
  printf("rod2= %d, rod1=%d, rod2 - rod1=%d\n",rod2,rod1, rod2-rod1);
    }
```

<u>Results</u>:

```
rod2=180,   rod1=102,  rod2 - rod1=39
```

Arithmetic operations with pointers allow you to use pointers to reference array elements instead of indexing them. For example, instead of A[i] you can use *(A + i). This code is more efficient and faster, although, probably, not so clear.

ARRAYS

An array is a collection of elements of a certain data type with a name. Each array element is defined by the array name and by the number (index) in the sequence of elements. Array index in C/C++ language is an integer; **the first element's index is always 0**.

The array size is defined by a constant or constant expression. A variable-size array can only be defined in the dynamic memory.

Definition:
type <array name> [dimension1] [dimension2] …[dimensionN]

Example: *int a [100]* – array of 100 elements from a[0] to a[99]

An array in memory always has a continuous memory space. Its size in bytes can be calculated as:

<base-type size> * <number of elements>

For example:

```
#include <stdio.h>
main() {
  int B[3];
  B[0] = 0;
  B[1] = 10; B[2] = 20;
  printf("B[0]= %d\n",B[0]);
  printf("B[1]= %d\n",B[1]);
  printf("B[2]= %d\n",B[2]);
}
```

Multi-dimentional arrays.

Examples:
int A[3][2]; //two-dimensional array of integers, 3 rows and 2 columns
char A[3][3][3][3]; // a four-dimensional symbolic array

Working with a two-dimensional array:

```
#include <stdio.h>
main() {
float B[4][3];
B[0][0] = 0;    B[1][1] = 1.1;    B[2][1] = 1.2;   B[3][1] = 1.3;
B[1][2] =  2.1;       B[2][2] =    2.2;   B[3][2] =    2.3;
printf("B[1,1]=     %4.2f     B[2,1]=     %4.2f     B[3,1]=     %4.2f\n",
B[1][1],B[2][1],B[3][1]);
   printf("B[1,2]=      %4.2f      B[2,2]=      %4.2f      B[3,2]=      %4.2f\n",
B[1][2],B[2][2],B[3][2]);
}
```

POINTERS AND ARRAYS

Arithmetic operations with pointers allow you to use pointers to reference array elements instead of indexing elements. The array name is a pointer to the array, i.e. to the first element of the array.

Since the index for the first array element is always **0**, then, if the array is described as follows

```
int a[10];
```

then ***a*** is equivalent to **&*a[0]*** and accordingly

a[3] is equivalent to **(a+3)*

a+i is equivalent to &*a[i]*

For example, if a pointer is declared

```
int *an;
    an=a;        /* an points to a[0] */
    an++;        /* an points to a[1] */
    an++;        /*now an points to a[2] */
```

Example:

1) Traditional summation of array elements:

```
int a[100];
suma=0;
for (int i=0; i<100; i++) {
      sum+=a[i];
}
```

2) Summation of array elements with the use of pointers:

```
int a[100];
suma=0;
for (int *ptr=&a[0]; ptr<&a[99]+1; ptr++)  {
                sum+=*ptr;
}
```

EXAMPLES OF PROGRAMMS

1. Sorting of array.

Typical simple sorting algorithms consist of the following main steps:

- comparison of two elements of a data structure (array, list) and their ordering;
- exchange of two elements if needed;
- loops that ensure comparison and exchange of all elements of an array until the array is arranged in a required sequence of order (usually, either ascending or descending order).

Example 1. Bubble sort (swap with selection).

The smallest elements of the array ("the lightest") come to the surface, the hardest – go down (sink). The algorithm looks at the entire array from "bottom-up" and replaces the adjacent elements if the next (lower) element is less than the top one. This will bring the smallest element of the array to the surface. Next, this algorithm is repeated for other *n-1* elements.

```
#include <iostream>
using namespace std;

int main() {
   int n;
   cout << "Number of elements: "; cin >> n;
   int x[n];
   cout << "Enter the elements: " << endl;
   for (int i = 0; i < n; i++)
      cin >> x[i];
   for (int m = 0; m < n; m++)    //entered array values are mirrored
      cout << x[m] << " ";
   cout << endl;

   int temp;      //temporal variable to exchange values

    //loop is repeated while all elements are tested
   for (int i = 0 ; i < n; i++)
   {
      for (int j = 0 ; j < n - i - 1; j++) {
         if (x[j] > x[j+1]) {          //exchange of elements if needed
            temp = x[j];
            x[j] = x[j+1];
            x[j+1] = temp;
         }
      }
```

```
        //output of array elements after each iteration
        for (int m = 0; m < n; m++)
            cout << x[m] << " ";
        cout << endl;
    }
 return 0;
}
```