STEP computer ACADEMY

PROGRAMMING C

Lesson No. 12

Programming

C

Contents

1. Static and dynamic memory allocation

Static memory is a storage place of all global and static variables. Static memory variable are once declared and deleted after program termination.

Dynamic memory or free storage memory differ by the fact that the program should directly request memory for elements stored within this area and then to release the memory if it is not needed anymore. Working with dynamic memory the program can allow itself to find out the number of array elements at the execution stage.

2. Pointers

Pointer is a variable containing address of another variable. Pointers are widely used in language C. It happens partly because sometimes they give the only chance to express necessary action, and partly because they usually lead to more compact and efficient programs in comparison with those that can be obtained in other ways.

Since an index contains an object address, it gives an opportunity of «auxiliary» access to this object through the pointer. Assume that **x — variable**, for example, type **int**, and **px** is **index** creates by any other indicated method. Unary operation **&** gives an object address so that an operator:

```
px = &x;
```

Assigns address **x** to variable **px**; they say that **px** «indicates» to **x**. Operation & is applicable only to variables and elements of an array of the construction type:

```
&(x-1) и &3
```

are illegal. It is impossible to obtain an address of register variable.

Unary operation * considers its operand as a destination address and refers this address to extract the contents. Therefore, if **y** also has the type **int**, then:

```
y = *px;
```

Assigns to **y** the contents of what is indicated by **px**. Thus the sequence

```
px = &x;
y = *px;
```

assigns to **y** the same value as an operator:

```
y = x;
```

variables participating in all that should be described as:

```
int x, y;
int *px;
```

We have already met several, times the description of **x** and **y**. Pointer description:

```
int *px;
```

is new and should be considered as mnemonic; it says that a combination ***px** has type **int**. it means that if px appears within the context *px, then it is equal to the variable of type int. actual syntax of variable description imitates syntax of expressions where this variable can appear. This observation is useful in all cases connected with complex descriptions. For example:

```
double  atof(), *dp;
```

says that **atof()** and ***dp** have in the expressions values of type **double**. You should also notice that from this description follows that a pointer can only points at the particular object type.

Pointers can belong within the expressions. For example, if px points at the integer x, then ***px** can appear in any context where we can meet **x**. For example:

```
    y = *px + 1;//assigns to y the value bigger that value x by one;
    cout<< *px;//outputs current value x;
    d = sqrt((double) *px)//gets within d square root of x,
/*wherein before transfer of the function sqrt value x transforms  into type double */
```

In the following expressions:

```
y = *px + 1;
```

unary operations * and **&** are bind with its operand tougher that arithmetic operations, so such an expression takes the value the **px** points at, adds 1 and assigns the result to the variable **y**. we will soon get back to the meaning of the expression:

```
y = *(px + 1);
```

References to indexes can appear also within the left part of assignations. If **px** points at **x**, then::

```
px = 0;
```

assumes **x** is equal to zero and:

```
*px += 1;
```

increases it by one just like the expression:

```
(*px) + 1;
```

Parenthesis are necessary in the last example; if we drop them out, since unary operations like * and ++ are executed from the right to the left, this expression increases **px** and not the variable it points at.

And at last, since pointers are variables we can handle them as with the rest of the variables. If **py** — is another pointer to the variable of type **int** then:

```
py = px;
```

copies the contents of px into **py**, in the result py points at the same as **px**.

3. Pointers and arrays

In language C there is a strong connection between the pointers and arrays, it is so strong that pointers and arrays should indeed be reviewed together. Any operation that can be exercised by means of array indexes, we can do it using the pointers. Variant with the pointers is quicker through a bit difficult for understanding, at least for a beginner. Description:

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

determines an array of size 10, i.e. a set of 10 sequential objects that called **a[0], a[1], ..., a[9]**. Record **a[i]** corresponds to array element through i positions from the beginning. If pa is an indicator of the integer described as:

```
int *pa;
```

then assignation:

```
pa = &a[0]
```

Leads to the fact that **pa** points at a zero element of array **a**. It means that pa contains an address of element **a[0]**. Now assignation:

```
x = *pa
```

will copy the contents **a[0]** into **x**.
If **pa** points at some particular element of array **a**, by definition

pa+1 points at the next element and **pa-i** points at the element standing on **i** positions before the element pointed at by pa, and **pa+i** points at the element standing on **i** positions after. Therefore, if **pa** points at **a[0]**, then:

```
*(pa+1)
```

refers to the contents of **a[1], pa+i** — address **a[i]**, and ***(pa+i)** — is the contents of **a[i]**.

These observations are true regardless the type of variables within the array **a**. The sense of definition «adding of 1 to a pointer», as well as its extension to include the pointers' arithmetic implies that assigning is scaled by the memory size that object holds and that a pointer points at. Therefore, **i** in **pa+i** before adding multiplies by an object sizes **pa** points at.

Obviously there is a close correspondence between indexing and pointer arithmetic. In reality a compiler converts a reference to array into a pointer at the array head. In the result array name is n indicating expression. From this follows a few useful consequences. Since an array name is a synonym of its zero element position then assigning:

```
pa = &a[0]
```

can be written **as pa = a**.

What is more surprising, at least at first sight, is that reference to **a[i]** can be written as ***(a+i)**. Upon analyzing an expression **a[i]** in language C it slowly converts into the type ***(a+i)**; these two forms are equal. If we apply an operation & to both parts of this equality relation then we get that &a[i] and a+i re also

identical: a+i — address of i element from the beginning a. from the other side, if pa is a pointer then it can be used in the expressions with the index: **pa[i]** is equal to ***(pa+i)**. to be precise, any expression including arrays and indexes can be written with the pointers and displacements and vice versa, wherein they can be met in one assertion.

There is one difference between an array name and pointer that we should take into account. Pointer is a variable and operations **pa=a** and **pa++** have sense. But array name is a constant and not variable: constructions of type **a=pa** or **a++**, or **p=&a** will be illegal.

When array name is transmitted to a function, in fact a position of an array head will be transmitted. Inside the called function such an argument is the same ad any other, so that an array name in the capacity of the argument is a pointer, i.e. variable containing address.

```
/* displays an array m */
  void ShowElements(int *m, int size)
  {
      int n;
      for (n = 0; n < size; m++,n++)
              cout<<*m<<"\t";

  }
```

Operation of increasing m is absolutely legal, for this variable is a pointer, **m++** does not affect the array addressed the **ShowElements** functions, but only increases a local address copy for the function ShowElements.

Description of formal parameters in the function definition like:

```
int m[];
```

and

```
int *m;
```

are equivalent; the type of description we should prefer is mainly determined by the fact what expressions are used when writing a function. If an array name is transmitted to a function, then depending on the convenience we assume that function operates either with an array or pointer; and we should act further in the relevant way. We can even use both operation types if it is appropriate and evident.

4. Pointers function arguments. Arguments passing by pointers

Since in C arguments passing to functions is exercised «by value», the called procedure has no direct possibility to change a variable from the calling program. What to do if you have to change the argument? For example, a sort program wants to change two order-breaking elements using the function with the name **swap**. With this purpose it is enough to write:

```
swap(a, b);
```

Defining a function swap the following way:

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

Due to a call by value **swap** cannot affect the argument **a** and **b** within the calling function.

Fortunately, there is a possibility to get a desired effect. Calling program transmits pointers which values should be changed:

```
swap(&a, &b);
```

Since an operation **&** gives a variable address then **&a** is a pointer at **a**. Within the very swap arguments are described as pointers and an access to actual data is carried on through them:

```
void swap(px, py)
  {
       int tmp;

      temp = *px;
      *px = *py;
      *py = temp;
  }
```

5. Home assignment

1. Given an array of integers. Using the pointers interchange array elements with even and odd indexes (i.e. array elements that take even places should be changed with the elements that take odd places).

2. Given two arrays arranged in an ascending order: A[n] and B[m]. Form an array C[n+m] consisting of elements of the arrays A and B arranged in an ascending order.

3. Given two arrays: A[n] and B[m]. We need to create a third array that should contain:

- Elements of both arrays;
- Elements common for two arrays;
- Elements of array A that are not included within B;
- Elements of array B that are not included within A;
- Elements of arrays A and B that are not common for them (in other words an aggregation of the results of previous varians).