STEP computer ACADEMY

PROGRAMMING C

Lesson No. 15

Programming in C

Contents

1. Multidimensional dynamic arrays

Back in action! We have already dealt with dynamic arrays, however, we would like to dip into this subject again and tell you something about the creation of multidimensional dynamic arrays.

A multidimensional array in C is actually one-dimensional array. New and delete operations allow creating and deleting dynamic arrays maintaining the illusion of an arbitrary dimensionality. Organization of dynamic array requires additional attention that is compensated with an important advantage: characteristics of the array (operands of the new operation) might not be constant expressions. This allows you to create multidimensional dynamic arrays of arbitrary configuration. The following example demonstrates working with the dynamic arrays.

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

 void main()
 {
        int i, j;

         // Variables used to describe the characteristics of arrays.
         int m1 = 5, m2 = 5;

         /*
         Organization of two-dimensional dynamic array is performed in two
stages. At first, one-dimensional array of pointers is created. Then every element
of this array is assigned with the address. Constant expressions are not needed to
characterize the size of arrays.
         */
         int **pArr = new int*[m1];
         for (i = 0; i < m1; i++)
                pArr[i] = new int[m2];

         pArr[3][3] = 100;
         cout << pArr[3][3] << "\n";



        //Sequential deletion of two-dimensional array…

         for (i = 0; i < m1; i++)
                delete[]pArr[i];
         delete[]pArr;
}
```

2. Examples of multidimensional dynamic arrays

Example 1. Organization of «triangle» two-dimensional dynamic array.

At first, one-dimensional array of pointers is created. Then all the elements of array are assigned with the address of one-dimensional array. The size (the number of elements) of every new array is greater or less by one than the size of the previous one. It is easy to do this by means of a variable within the square brackets that is an operand of the new operation.

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

void main()
{
        int i, j;

        // The variables used to describe the characteristics of arrays.
        int m1 = 5, wm = 5;
        int **pXArr = new int*[m1];

        for (i = 0; i < m1; i++, wm--)
                pXArr[i] = new int[m1];


        //Filling an array with zero and displaying it
        for (i = m1 - 1; i >= 0; i--, wm++) {
                for (j = 0; j < wm; j++){
                        pXArr[i][j]=0;
                        cout<<pXArr[i][j]<<"\t";
                }
                cout<<"\n\n";
        }

    /* Sequential deletion of two-dimensional array of triangular configuration  */
```

```
        for (i = 0; i < m1; i++)
                delete[]pXArr[i];
        delete[]pXArr;
}
```

Example 2. Organization of three-dimensional dynamic array.

Creating and destroying a three-dimensional array requires an additional iteration. However, there is also nothing new.

```
#include <iostream>
using namespace std;
void main()
{
        int i, j;

        // The variables used to describe the characteristics of arrays.
        int m1 = 5, m2 = 5, m3 = 2;

        // a pointer to a pointer to a pointer:)
        int ***ppArr;

        // Creating an array
        ppArr = new int**[m1];
        for (i = 0; i <m1; i++)
                ppArr[i] = new int*[m2];

        for (i = 0; i < m1; i++)
                for (j = 0; j < m2; j++)
                        ppArr[i][j] = new int[m3];

        ppArr[1][2][3] = 750;
        cout << ppArr[1][2][3] << "\n";

        // Deletion according to a sequence that is opposite to creation
        for (i = 0; i < m1; i++)
                for (j = 0; j < m2; j++)
                        delete[]ppArr[i][j];

        for (i = 0; i < m1; i++)
                delete[]ppArr[i];
        delete[] ppArr;
}
```

3. Enumeration types

Enumeration type is introduced by means of the «enum» keyword. It defines a set of values specified by a user. The set of values is enclosed in curly braces and it is a whole set of named constants represented by their identifiers. These constants are called enumerated constants. Let's consider the following declaration:

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

By means of this declaration it is possible to create an integer type of four kinds of suits referring to integer constants. Enumerated constants are the CLUBS, DIAMONDS, HEARTS and SPADES identifiers having the values: 0, 1, 2 and 3, respectively. These values are assigned by default. The first enumerated constant is assign with 0 (an integer constant numerical value). Each of the subsequent members of the list is greater by one than the member arranged at the left. Variables of the Suit type defined by the user can be assigned to only one of four values declared in the enumeration.

Another popular example of the enumerated type:

```
enum Months {JAN = 1, FEB, MAR, APR, MAY, YUN, YUL AUG, SEP, OCT, NOV, DEC};
```

This declaration creates a user-defined type called Months with enumerated constants representing the months of year. Since the first value of the given enumeration is set to 1, the remaining values from 1 to 12 are increased by 1.

In declaration of enumerated type any enumerated constant can be assigned to an integer value.

Note: A typical mistake. After the enumerated constant is defined, an attempt to assign a different value to it is a syntax error.

Basic peculiarities of using enumerations.

1. Using enumerations instead of integer constants facilitates reading the program.

2. enum identifiers must be unique, but some of enumeration constants may have the same integer values.

3. A set of identifiers of enumeration type is a unique type differing from the other integer types.

4. Enumerated constants can be defined and initialized by arbitrary integer constants and constant expressions:

```
enum ages {milton = 47, ira, harold = 56, philip = harold + 7};
```

Note: Please note that when there is no definite initializer, the default rule is applied, so – ira = 48. In addition, the values of enumerated constants may not be unique.

5. Each enumeration is a separate type.

The type of enumerator is an enumeration itself. For example, in

```
enum Keyword {ASM, AUTO, BREAK};
```

AUTO имеет тип Keyword.

6. Enumerated constant can be declared anonymously, i.e. without the name of type.

```
enum {FALSE, TRUE};
enum {lazy, hazy, crazy} why;
```

First declaration is a common method of declaring mnemonic integer constants. The why variable of enumerated type is secondary. Permissible values of this variable are lazy, hazy and crazy.

7. Enumerations may be implicitly converted to common integer types, but not vice versa.

```
enum boolean {FALSE, TRUE} q;
enum signal {off, on} a = on;  //a initialized into on
enum answer {no, yes, maybe = -1} b;

int i, j = true;  //true is converted into 1

a = off;  //true

i = a;  //true i becomes 1

q = a;  //false two different types

q = (boolean)a;  //true explicit conversion by means of cast
```

4. Pointers to functions

Before introducing a pointer to function, it is to be recalled that every function is characterized by the type of returned value, name and signature. Signature is determined by the number, order and types of parameters. Sometimes, it is said that the signature of function is a list of types of its parameters.

Now we are going to discuss the theme of this section of the lesson using a sequence of statements.

1. Using the name of function without subsequent brackets and parameters, the name of function appears for a pointer to this function, and its value is the address of function in the memory.

2. This value of address may be assigned to a pointer, and then this new pointer may be used to call a function.

3. Determination of a new pointer must involve similar type as the value returned by function, and similar signature.

4. A pointer to function is defined as follows:

```
function_type(*pointer_name)(parameter_specification);
```

For example: int (*func1Ptr) (char); is a definition of the func1Ptr pointer to a function with char type parameter returning the value of int type.

Note: Be careful!!! If the above syntax construction is written without first parentheses, i.e. in the form of int *fun (char);, it will be considered by the compiler as a prototype of function named fun and having a parameter of the char type that returns the value of pointer of the int * type.

A second example: char * (*func2Ptr) (char * ,int) is a definition of the func2Ptr pointer to a function with such parameters as pointer to char and int type returning the value of a pointer to char type.

Let us show it in practice.

When defining a pointer to function, the type of returned value and signature (types, number and order of parameters) must match the corresponding types and signatures of functions, the addresses of which are supposed to be assigned to the introduced pointer when initializing or using the assignment operator. In order to demonstrate the above said let's consider a program with a pointer to function:

```
#include <iostream>
using namespace std;
void f1(void)        // Definition f1.
{
        cout << "Load f1()\n";
 }
void f2(void)        // Definition f2.
{
         cout << "Load f1()\n";
}
void main()
{
        void (*ptr)(void);  // ptr is a pointer to function.
        ptr = f2;               // The assigned address is f2().
        (*ptr)();               // Calling f2() by its address.
        ptr = f1;               // The assigned address is f1().
        (*ptr)();               // Calling f1() by its address.
        ptr();                  // The call is equivalent to (*ptr)();
}
Program output:

        Load f2()
        Load f1()
        Load f1()
        Press any key to continue
```

Here, the value of the pointer_name is the address of function. Calling this function by address is provided by means of the dereferencing operator *. However, it would be a mistake to write the function call as *ptr() without parentheses. The fact is that the () operation has a higher priority than the operation of calling by address *. Consequently, in accordance with the syntax we will try to call the ptr() function. And dereference operation will be assigned to the result and interpreted as a syntax error.

In the process of definition a pointer to function can be initialized. The function address should be used as an initialization value. The type and signature of this function correspond to the determined pointer.

When assigning pointers to functions, it is also necessary to observe the correspondence of types of the returned values of functions and signatures for the pointers of the left and right parts of the assignment operator. The same is true relating to the subsequent calls of functions using pointers, i.e. the types and number of actual parameters used when calling the function by the address must correspond to the formal parameters of the called function. For example, only some of these operators will be adopted:

```
char f1(char) {...}       // Function definition.
char f2(int) {...}        // Function definition.
void f3(float) {...}      // Function definition.
int* f4(char *){...}      // Function definition.
char (*pt1)(int);         // A pointer to function.
char (*pt2)(int);         // A pointer to function.
void (*ptr3)(float) = f3; // Initialized pointer.
void main()
{
    pt1 = f1;  // Error - signature mismatch.
    pt2 = f3;  // Error - type mismatch (values and signatures).
    pt1 = f4;  // Error - type mismatch.
    pt1 = f2;  // Correct.
    pt2 = pt1; // Correct.
    char c = (*pt1)(44); // Correct.
    c = (*pt2)('\t');    // Error - signature mismatch.
}
```

The following program reflects the flexibility of mechanism of calling the functions using pointers.

```
#include <iostream>
using namespace std;
// Functions of the same type with similar signatures:
int add(int n, int m) { return n + m; }
int division(int n, int m) { return n/m; }
int mult(int n, int m) { return n * m; }
int subt(int n, int m) { return n - m; }
void main()
{
        int (*par)(int, int); // A pointer to function.
        int a =  6, b = 2;
        char c = '+';
        while   (c != ' ')
        {
                cout << "\n Arguments: a = " << a <<", b = " << b;
                cout << ". Result for c = \'" << c << "\':";

                switch (c)
                {
                case '+':
                        par = add;
                        c = '/';
                        break;
                case '-':
                        par = subt;
                        c = ' ';
                        break;
                case '*':
```

```
                        par = mult;
                        c = '-';
                        break;
                case '/':
                        par = division;
                        c = '*';
                        break;
                }
                cout << (a = (*par)(a,b))<<"\n"; //Calling by address.
        }
}
Program output:


        Arguments: a = 6, b = 2. Result for c = '+':8

        Arguments: a = 8, b = 2. Result for c = '/':4

        Arguments: a = 4, b = 2. Result for c = '*':8

        Arguments: a = 8, b = 2. Result for c = '-':6
        Press any key to continue
```

The cycle is continued until the value of «c» variable will become a gap. In every iteration the par pointer receives the address of one of the functions and the «c» value is changed. Considering the results of program it is easy to follow the order of execution of its operators.

Arrays of pointers to functions.

Pointers to functions may be combined into arrays. For example, float (*ptrArray[4]) (char); is a description of an array named ptrArray including four pointers to functions. Each of these functions has a parameter of the char type and returns a value of the float type. For example, in order to address the third of these functions, you will require the following operator:

```
float a = (*ptrArray[2])('f');
```

Array indexing starts with 0, and thus, the third element of the array has an index of 2.

Arrays of pointers to functions are useful when developing different menus, i.e. the programs managed via the menus. For this purpose the actions offered the future user of the program are formed as functions. The addresses of these functions are placed into the array of pointers to functions. The user is asked to choose the required item (usually he/she enters the number of item being selected) from the menu. The address of function is selected from the array by the number of item. This way is similar to the one when the index is used. Calling a function by this address ensures completion of the required actions. The most generalized scheme of implementation of this approach is shown by the following «file processing» program:

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

/* Defining the functions for menu processing
(functions are fictional, i.e. they do not execute real actions):*/

void act1 (char* name)
{
        cout <<"Create file..." << name;
}
void act2 (char* name)
{
        cout << "Delete file... " << name;
}
void act3 (char* name)
{
        cout << "Read file... " << name;
}
void act4 (char* name)
{
        cout << "Mode file... " << name;
}
void act5 (char* name)
{
        cout << "Close file... " << name;
}
```

```
void main()
{
        // Creating and initializing the array of pointers
        void (*MenuAct[5])(char*)  = {act1, act2, act3, act4, act5};

        int number;  // The number of the chosen menu item.
        char FileName[30];  // A line for file name.

        // Menu implementation
        cout << "\n 1 - Create";
        cout << "\n 2 - Delete";
        cout << "\n 3 - Read";
        cout << "\n 4 - Mode";
        cout << "\n 5 - Close";

        while (1)  // Infinite loop.
        {
                while (1)
        { /* Loop will be continued until a correct number will be entered.*/
                        cout << "\n\nSelect action: ";
                        cin >> number;
                        if (number>>= 1 && number <= 5) break;

                        cout << "\nError number!";
                }
                if (number != 5)
                {
                        cout << "Enter file name: ";
                        cin >> FileName; // Read the file name.
                }
                else break;
                // Calling a function by a pointer to it:
                (*MenuAct[number-1])(FileName);
        } // End of the infinite loop.
}
```

The menu items are repeated until the number 5 is entered (closing).

5. Home assignment

1. Write a program adding a row or a column into any place of the two-dimensional matrix by the user's choice.

2. The matrix of MxN order is given (M stands for rows, N — columns). It is necessary to fill it with values and write a function performing a cyclic shift of the lines and/or columns of the array a specified number of times in the specified direction.