STEP computer ACADEMY

PROGRAMMING C

Lesson No. 14

Programming in C

Contents

1. Strings in C ++

String array declaration syntax. Initialization.

In previous lessons, you explored different types of arrays. Today we will examine another type of array, a string array, because this type is worthy of a special attention. The strings are intended for input, processing and output of symbolic information.

A **string constant** is a sequence of zero or more characters enclosed in quotation marks. The quotes are not a part of a string constant and serve only for its limiting.

Strings are represented as an array of char elements. It means that the string characters can be represented as disposed in adjacent memory cells — a character per cell. However, an array of characters is not always a string!

Consider the following line as an example: «Character string.

The quotes are not a part of the string. They are introduced in order to mark its beginning and end, that is, i.e. play the same role as the apostrophes with a single character (each cell — 1 byte).

S	Y	M	B	O	L		S	T	R	I	N	G	\0

It should be noted that the character '\ 0' is the last element of the array. This is a zero-character: it is used to mark the end of the string in C. A zero- character is not the number 0; it is not printed and has the number 0 in the table of ASCII codes.

A zero-character means that the number of array cells must be at least one bigger than the number of characters that must be placed in memory.

Do not confuse a character constant with a string containing a single character: 'X' is not the same thing as "X". In the first case it is a single character. In the second case it is a string of one character (the letter X) and a symbol of the end of the string '\ 0'.

Principles of string array initialization.

Here is a sample of the string array initialization:

```
#include <iostream>
using namespace std;
int n=5;
// string array initialization.

char line[5] = { 'C','a','t','!','\0' };
void main ()
{
       cout << "Word: ";
       for (int i=0; i<n; i++)
          cout << line[i];
}
```

The above example is not very convenient for creating long strings. Moreover, the string literal output looks rather odd in the cycle, isn't it? A a special type of initialization is designed for character arrays. Instead of braces and commas, a character string enclosed in quotation marks can be used. It is not necessarily to set the array size, because the compiler itself

defines the length by counting the number of initial values.

The output operation cout is designed so as to only specify the name of a string array in order to immediately display it on the screen.

```
#include <iostream>
using namespace std;
int n=5;
char line[] = "Cat!"; // string array initialization.

void main ()
{
      cout << "Word: ";
      // Display the string array a on the screen.
      cout <<line;
}
```

2. Strings and pointers. Interaction

We discussed pointers in previous lessons, so you must remember how closely they are interrelated with arrays. Today we will revive our knowledge about pointers. Access to the string in the program is performed by means of a character pointer. If you define the variable message as:

```
char *message;
```

then after performing the statement

```
message = "and bye!";
```

message will become a string pointer. Note that the statement cin cannot be applied to this pointer.

```
#include <iostream>
using namespace std;
char *message;
char privet[] = "and bye!";
char *pr = Hello;
void main ()
{
      message = "Hello";
      cout << " " << message << " " << pr << "\n";
      int i = 0;
      while  (*(pr+i)!='\0')
      {
         cout<< *(pr+i++) << " ";
      }
}
```

Pointers are often used in the string arrays. In such case, each string can be addressed by a pointer to its first character. It is convenient because rearranging pointers in the array of pointers and not the strings is enough for permutation of two strings arranged in the incorrect order, in fact.

We consider the function **month_name ()**, which returns a pointer to a string containing the name of the n-th month. This is a typical problem for a string array.

The function **month_name ()** contains a local string array and returns a pointer to the required string when accessing it.

In the description of an array of pointers to the character **name []**, a simple list of strings is an initializer. Characters of the i-th string are located in a specific memory location, while a pointer to its beginning is stored in the element name [i]. Since the size of the array name is not specified, the compiler counts the initializers and establishes the correct number.

```
#include <iostream>
using namespace std;
const int n=15;
void main ()
{
      char *month_name(int);
      /* ------------- */
      for (int i=0; i < n; i++)
         cout << "Month number " << i << " - " <<  month_name(i) <<"\n";
 }
/* ------------------------------------------ */
char *month_name (int k)  /* Name of the k-th month */
{
      static char *name[] = {
                              "none","January",
                              "February","March","April",
                              "May","June","July","August",
                              "September","October","November",
                              "December"
                             };
      return (k<1||k>12)?name[0]:name[k];
}
```

Program performance result:

```
Month number 0 - none
Month number 1 - January
Month number 2 - February
Month number 3 - March
Month number 4 - April
Month number 5 - May
Month number 6 - June
Month number 7 - July
Month number 8 - August
Month number 9 - September
Month number 10 - October
Month number 11 - November
Month number 12 - December
Month number 13 - none
Month number 14 - none
Press any key to continue
```

3. String library. String functions

Below we list the main string functions. Most of these functions prototypes (if not stated apart) are stored in the header file string.h.

int getchar(); — returns the value of the character (if any) that the user has typed on the keyboard. After entering the character you need to press Enter. The header file — stdio.h

int getch(); — same as above, but the character is not displayed on the screen. Most often it is used for program performance delay execution. The header file — **conio.h**

int putchar(int c); — displays the character c on the screen. If successful, it returns the character c, if not — **EOF**.

The header file — **stdio.h**

char *gets(char *s); — reads characters, including spaces and tabs, until it meets a newline character that is replaced by a null character. The sequence of the characters read is stored in the memory area addressed by the argument s. If successful, it returns the argument s, in case of error — zero.

The header file — **stdio.h**

int puts(const char * s); — displays a string defined by the argument const char *s. The header file — stdio.h

char *strcat(char *dest, const char *scr); — combines the intial string scr and the resulting string dest, attaching the former to the latter. **Returns dest**.

char *strncat (char *dest, const char *scr, int maxlen); — combines maxlen of the characters of the initial string scr and

the resulting string dest, attaching a part of the former to the latter. **Returns dest**.

char *strchr (const char *s, int c); — searches the first occurrence of the character c in the string s, starting from the beginning of the string. If successful, it returns a pointer to the found character, otherwise it returns zero.

char *strrchr (const char *s, int c); — same as above, but the search is carried out starting from the end of the string.

int strcmp(const char *s1, const char *s2); — compares two strings. It returns a negative value if s1<s2; it returns zero if s1 == s2. The positive value is returned if s1> s2. Parameters are pointers to the strings being compared.

int stricmp(const char *s1, const char *s2); — same as above, but the comparison is case-insensitive.

int strcmp(const char *s1, const char* s2, int maxlen); — same as above, but only the first maxlen characters are compared.

int strnicmp (const char s1, const char *s2, int maxlen); — same as above, only the first maxlen characters but the comparison is case-insensitive.

int strcspn (const char *s1, const char *s2); — returns the length of the maximal initial substring of the string s1, which does not contain characters from the second string s2.

int strlen(const char *s); — returns the length of the string s — the number of characters preceding the null character.

char *strlwr (char *s); — converts all uppercase (capital) letters in lowercase (small) letters in the string s.

char *strupr (char *s); — converts all lowercase (small) letters in uppercase (capital) letters in the string s.

char *strnset(char *s, int c, int n); — fills the string s with characters c. The parameter n specifies the number of characters placed in a string.

char *strpbrk(const char *s1, const char *s2); — searches the first occurrence of any character from a string s2 in the string s1. It returns a pointer to the first found character or zero, if the character is not found.

char *strrev (char *s); — reverses the sequence of characters in the string (except the terminating null character). The function returns the string s.

char *strset (char *s, int c); — replaces all the characters of the string s with the given character symbol c.

i**nt strspn(const char *s1, const char *s2);** — calculates the maximum length of the initial substring of the string s1, containing only characters from the string s2.

char *strstr (const char *s1, const char *s2); — searches the string s2 in the string s1. It returns the address of the first character of the string s2 occurrence. It returns zero, if the string is missing

char *strtok(char *s1, const char *s2); — divides the initial string s1 into tokens (substrings) separated by one or more characters from the string s2.

double atof(const char *s); — converts the string s to a floating point number of the type double. The header file — math.h

int atoi(const char *s); — converts the string s to the number of the type int. It returns a value or zero, if the string cannot be converted. The header file — stdlib.h

long atol(const char *s); — converts the string s to the number of the type long. It returns a value or zero, if the string cannot be converted. The header file — stdlib.h

char *itoa(int value, char *s, int radix); — converts the integer value to the string s. It returns a pointer to the resulting string. The value radix is a numeric base used in conversion (2 to 36). The header file — stdlib.h

4. Strings in C. Samples

The previous topic of the lesson was devoted to the string functions, and today's topic focuses on their application.

Definition of the string length.

The string length is simply defined. To do this, pass the pointer of the function **strlen ()**, which returns the length of the string expressed in characters. After the declaration

```
char *c = "Any old string....";
int len;
```

the next statement sets the variable len equal to the length of the string addressed by a pointer **c**:

```
len = strlen(c);
```

Function strlen(). Usage sample.

```
#include <stdio.h>
#include <string.h>
#include <iostream>
using namespace std;
const int MAXLEN=256;
void main()
{
        char string[MAXLEN]; /* Place for 255 characters. */
        cout << "Input string:: ";
        gets(string);
        cout <<"\n";/* Start a new line. */
        cout << "String: " << string << "\n";
        cout << "Length = " << strlen(string);
}
```

A **string** variable string for receiving input from the function **gets()** is defined here. Once a string is entered, the program gives the variable **string** of the function **strlen()**, which calculates the length of the string in characters.

Other types of strings can be also given the function **strlen ()**. For example, you can define and initialize a character buffer as follows:

```
char buffer[128] = "Copy in buffer";
```

Use the function strlen()to set the integer variable len, equal to the number of characters in the literal string copied to the clipboard:

```
int len;        /* Define the integer variable. */
len = strlen(buffer); /* /* Compute the string length. */
```

Copying strings.

The assignment operator is not defined for strings. If c1 and c2 are character arrays, you cannot copy one to another as follows:

```
c1 = c2; //???
```

If **c1** and **c2** are defined pointers of the type **char***, the compiler would agree with the statement, but you will unlikely get the expected result. Instead of copying the characters from the string to another, the statement **c1 = c2 copies the pointer c2 to the pointer c1**, thus overwriting the address into **c1**, potentially losing the information addressed to by the pointer.

```
char*c1 = new char [10];
char*c2 = new char [10];
c1=c2;// memory allocated for c1 is lost
```

To copy one string into another, call the function of copying strings **strcpy()**, instead of using the assignment operator. For two pointers **c1** and **c2** of the type **char***, the statement

```
strcpy(c1, c2);
```

copies the characters addressed to by the pointer **c2** into the memory addressed to by the pointer **c1**, including terminating zeros. Only you are responsible for enough place for storing the copy left by the receiving string.

The same function **strncpy()** limits the number of characters to be copied. If the **source** and the **destination** are pointers of the type **char*** or character arrays, the statement

```
strncpy(destination, source, 10);
```

copies up to 10 characters from the string addressed to by the pointer **source**, to the memory location addressed to by the pointer **destination**. If the string source has more than 10 characters, the result is truncated. If less — unused bytes of the result are set equal to zero.

Note: String functions, whose name contains an additional letter n, declare a numeric parameter, limiting action of the function. These features are more secure, but slower than their analogs that do not contain the letter n. The program samples include the following pairs of functions: **strcpy() and strncpy(), strcat() and strncat(), strcmp() and strncmp()**.

String concatenation

Concatenation of two strings means their coupling, which creates a new, longer string. When you declare a string

```
char original[128] = "Test ";
```

the statement

```
strcat(original, " one, two, three!");
```

converts the value of the intiail string original to «Test one, two, three!»

When you call the function **strcat()**, make sure that the first argument of the type **char*** is initialized and has enough space to store the result. If **c1** addresses the string that is already filled, and **c2** addresses a non-zero string, the statement **strcat (c1, c2);** overwrites the end of the string, causing a fatal error.

The function **strcat()** returns the address of the resulting string (which coincides with its first parameter) and can be used as a cascade of multiple function calls:

```
strcat(strcat(c1,c2),c3)
```

The following sample shows how to use the function strcat() to get the first and the last names, which are stored separately, for example in the form of database fields, in a single string. Enter the first and the last names.

The program will concatenate the entered strings and display them as a separate string.

```
#include <iostream>
#include <string.h>
using namespace std;
void main()
{
        //Reserve space to enter two strings.
        char *fam = new char[128];
        char *im = new char[128];
        char *otch = new char[128];
        //Data input.
        cout << "Enter" << "\n";
        cout << "\tSurname: ";
        cin >> fam;
        cout << "\tName: ";
        cin >> im;
        cout << "\tLastname: ";
        cin >> otch;
        //Reserve space for the result.
        //It is necessary to take into account two spaces and the resulting
        //null character.
        char *rez=new char[strlen(fam)+strlen(im)+strlen(otch)+3];
        //"Assembling" the result.
        strcat(strcat(strcpy(rez,fam)," "),im);
        strcat(strcat(rez," "),otch);
        //Return the memory heap.
        delete [] fam;
        delete [] im;
        delete [] otch;
        //Return the result.
        cout << "\nResult: " << rez;
        delete [] rez;
}
```

The above program demonstrates an important principle of the string concatenation: the first string argument must be always initialized. In this case, a character array **res** is initialized by calling the function **strcpy()**, which inserts **lastname** into **res**. The program adds blank spaces and the other string — **name**. You should not call the function strcat() with the uninitialized first argument.

If you are not sure that there is enough space for the substrings being connected, call **strncat()**, which is similar to the function **strcat()**, but requires a numeric argument specifying the

number of the characters being copied. For the strings **s1** and **s2**, which can be either pointers of the type **char***, or symbolic arrays, the statement

```
strncat(s1, s2, 4);
```

connects a maximum of four characters from **s2** to the end of the string **s1**. The result is necessarily null-terminated.

There is one way to use the function **strncat()**, which guarantees secure concatenation. It consists in transfer of the size of free memory of the destination string as the third argument to the function strncat().

Let us consider the following declarations:

```
const int MAXLEN=128
char s1[MAXLEN] = "Cat";
char s2[] = "in hat";
```

You can attach **s2** to **s1**, forming the string **«Cat in hat»** by means of **strcat()**:

```
strcat(s1, s2);
```

If you are not sure that there is enough space to store the result in **s1**, use an alternative statement:

```
strncat(s1, s2, (MAXLEN-1)-strlen(s1));
```

This method ensures that **s1** is not overflown even if **s2** needs to be cut to a suitable size. This statement works fine if s1 is a null string.

Programs often search for individual characters or substrings in the strings, especially when checking file names in the specified extension. For instance, after the user has been prompted to enter the file name, it is checked whether he has entered the extension **.TXT**; if yes, the action different for the one set for the extension **.EXE** is performed.

You may also reject all extensions, except for a certain one, which will help prevent errors caused by loading data file of undesired types.

Searching characters.

Usage sample of the function **strchr()**.

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;
void main()
{
  char *filename = new char[128];
  cout << "Enter name of file: ";
  gets(filename);
  cout << "\nName of file: " << filename << "\n";
  if (strchr (filename,'.'))
     cout << "Name has extension" << "\n";
  else
     strcat (filename,".TXT");
  cout << "Name of file: " << filename << "\n";
  delete [] filename;
}
```

This program finds the extension in the file name, performing search of the point among characters of the input string. (There can be only one point, which must precede the extension if it is available, in the file name.) The key statement in this program is

```
if (strchr (filename,'.'))
     cout << "Name has extension" << "\n";
else
     strcat (filename,".TXT");
```

The expression strchr(filename, '.') returns a pointer to a point in the string addressed to by the pointer filename.

If the character is not found, the function strchr() returns zero. As non-zero values represent the "truth", you can use strchr() as a returning value "true" / "false". You can also use the function strchr() to assign the pointer to a substring starting at a given character. For example, if p is a pointer declared as char *, and the pointer filename addresses the string TEST. TXT, the result of the operator p= strchr(filename, '.'); is shown in Figure.

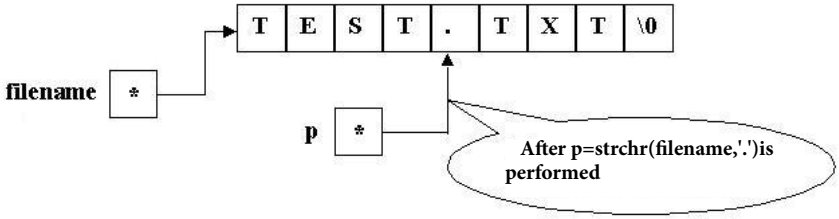

Figure shows another important moment related to the pointer addressing to a part of the string, i.e. substring, and not a full string. These pointers should be used with a great caution. Figure shows a single string, **TEST.TXT**, terminating with a null byte, but two pointers are indicated — **filename** and **p**. **Filename** addresses the full string. **P** addresses the substring within the same character set.

String functions do not care about the number of bytes that precede their first character. Therefore, the statement

```
puts(p);
```

displays the substring .TXT as if it is full string variable, not a part of the other string.

Nothing is unusual in using numerous pointers, which addresses substrings of the same string, in C programming. The string shown in Figure is in heap, so the statement

```
delete [] p;
```

thus trying to free the substring addressed to by the pointer p, which will undoubtedly lead to the destruction of the heap, causing an error related to the category of difficult-t-locate.

The function strchr() finds the first occurrence of a character in a string. Declarations and statements

```
char *p;
char s[]="Abracadabra";
p = strchr(s,'a');
```

assign address of the first lowercase letter 'a' in the string «Abracadabra» to the pointer **p**.

The function **strchr()** considers terminating null of the string as a significant symbol. Taking into account this fact, we can find the address of the string. Considering the previous declarations, the statement

```
p = strchr(s,0);
```

sets the pointer p equal to the address of the substring «bra» at the end of the string «Abracadabra».

Substring search.

Besides searching characters in a string, you can also hunt after substrings. The sample demonstrates this method. This program is similar to the previous one, but sets the file extension .**TXT**.

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;
void main()
{
  char *filename = new char[128],*p;
  cout << "Enter name of file: ";
  gets(filename);
  cout << "\nName of file: " << filename << "\n";
  strupr(filename);
  p = strstr (filename,".TXT");
  if (p)
     cout << "Name has extension" << "\n";
  else
   { p = strchr (filename,'.');
     if (p)
        *p=NULL; //Delete any other extension.
     strcat (filename,".TXT");
   }
  cout << "Name of file: " << filename << "\n";
  delete [] filename;
}
```

This program creates a file name that ends with the obligatory extension.TXT. To determine whether the file name includes this extension, the program executes the statement

```
p = strstr (filename,".TXT");
```

Like **strchr ()**, the function **strstr ()** returns the address of the substring or zero, if the search string is not found.

If the goal is detected, the pointer **p** becomes equal to its address; in this sample — to the point's address in the substring **.TXT**. Since the extension can be entered with lowercase letters, the program executes the statement

```
strupr(filename);
```

to convert the characters of the initial string to uppercase letters before calling **strstr ()**.

The sample also demonstrates how to truncate the string at the position of a given character or substring. The function strstr () is called here in order to set the pointer p equal to the address of the first point in the string **filename**. If the search result is not zero, then the statement, which replaces the point with zero byte, will be executed:

```
*p = NULL;
```

This will attach a new end of the string in a place where there previously was the file extension. Now the string is ready to add a new extension by calling **strcat ()**.

5. Problem samples

In this section, we have prepared a few intuitive usage samples with pointers.

The program for replacement all letters «a» on the combination «ky» in the word X.

```
#include <string.h>
#include <stdio.h>
void main ()
{
        /*
                k - variable to pass the initial array
                i - variable to pass the terminative array
                n - length of the initial array
        */
        int k=0,i=0, n;
        /*
                x1 - initial array
                x2 - effective array (two times larger, in case
                     if the initial array is completely filled with the letters 'a')
                px1 - pointer to pass through the initial array
                px2 - pointer to pass through the terminative array
        */
        char x1[40],x2[80],*px1,*px2;

        // Request for the original array
        puts( "Enter word (max 39 letters) ");
        gets(x1);

        /*
        write the address of the initial and terminative
        arrays into pointers
        */
        px1 = x1;
        px2 = x2;

        /*
        calculate the real length of the initial array
        */
        n = strlen(x1)+1;

        // the loop enumerates the length of the initial
        // array elementwise
        while  (k<n)
```

```
        {
                // If the value of the current element
                // does not match c 'a'
                if (*(px1+k)!='a')
                {
                        // Copy the current element
                        // to the terminative array
                        *(px2+i) = *(px1+k);
                        // move to the next elements
                        i++;
                        k++;
                }
                // If the value of the current element
                // matches c 'a'
                else
                {
                        // Write the character 'k' in the current
                        // position of the terminative array
                        *(px2+i) = 'k';
                        // Write the symbol 'y' to the next
                        // position of the terminative array
                        *(px2+i+1) = 'y';
                        // move to the next element
                        // of the initial array
                        k++;
                        // "Jump" through an element
                        // of the terminative array
                        i += 2;
                }
        }
        // Demonstrate the terminative array
        puts(x2);
}
```

The program to replace all combinations «ky» on the letter «a» in the word X.

```
#include <string.h>
#include <stdio.h>
void main ()
{
   /*
                k - variable to pass the initial array
                i - variable to pass the terminative array
                n - length of the initial array
        */
        int k=0,i=0, n;
        /*
```

```
        x1 - the initial array
        x2 - the terminative array
        px1 - pointer to pass through the initial array
        px2 - pointer to pass through the terminative array
*/
char x1[40],x2[40],*px1,*px2;

// Request for the initial array
puts( "Enter word (max 39 letters) ");
gets(x1);

/*
write the address of the initial and terminative
arrays into pointers

*/
px1 = x1;
px2 = x2;

/*
calculate the real length of the initial array
*/
n = strlen(x1)+1;

// the loop enumerates the length of the initial
// array elementwise
while (k<n)
{
        // check if two characters
        // in the current position of the initial
        // array does not coincide with the combination "ky"
        if (strncmp((px1+k),"ky",2)!=0)
        {
                // copy a character
                // from the current position
                // to the terminative array and
                // move to the character forward
                *(px2+i++) = *(px1+k++);
        }
        // if two characters
        // in the current position of the original
        // array coincide with the combination of letters "ky"
        else
        {
                // write the character 'a' in the terminative
                // array and move one character forward
                // move two characters forward in the initial array
                *(px2+i++) = 'a';
                k += 2;
        }
}
```

```
        // demonstrate the terminative array
        puts(x2);
}
```

Program, doubling every letter of the word X.

```
#include <string.h>
#include <stdio.h>
void main ()
{
        // n - the length of the initial array  *2
        int n;

        /*
                x1 - the initial array
                x2 - the terminative array (more than twice)
                px1 - the pointer to move through the initial array
                px2 - the pointer to move through the terminative array
        */
        char x1[40],x2[80],*px1,*px2;

        // Prompt the original array
        puts( "Enter word (max 39 letters) ");
        gets(x1);
        /*
        write the addresses of the initial and terminative arrays of pointers
        */
        px1 = x1;
        px2 = x2;
        /*
        calculate the double length of the terminative array
        */
        n = 2*strlen(x1);

        // write '\0' to the last element of
        // the terminative array  '\0'
        // Demonstrate the terminative array
    *(px2+n) = '\0';

        // the loop enumerates the length of the initial
        // array elementwise
    while  ((*px1)!='\0')
    {
                // write the value of the current position
                // of the initial array to the current position
                // of the terminative array, in the latter
                // move one element forward
                *px2++ = *px1;
                // write the value of the current position
```

```
                // of the initial array to the current position
                // of the terminative array to both arrays
                // move one element forward
                *px2++ = *px1++;
        }
            // Demonstrate the terminative array
            puts(x2);
}
```

6. Home assigment

1. The user enters a string in a fixed array from the keyboard. It is necessary to check how many array elements are used and how many of them are free.

2. Display characters from m to n of the string entered by the user and write the given range to another array (the user enters m and n).

3. Remove characters from m to n, overwrite the string and display it on the screen.

4. The user enters a separate string and a character, it is necessary to display only the number of the last match (numbering starts from one).