Computers Programming
Course 11

Iulian Năstac

Recap from previous course

Cap. **Matrices (Arrays)**

- Matrix representation is a method used by a computer language to store matrices of different dimension in memory.
- **In C programming language, a matrix is a collection of variables (of the same type) that can be called with the same name.**
- C programming language uses "Row Major" schema, which stores all the elements for a given row contiguously in memory.

Recap from previous course

Declaring Arrays

- Syntax:

  type arrayName [lim_1] [lim_2] … [lim_n];

- where **lim_i** is the limit of index ***i*** (on dimension ***i***)
- The index ***i*** can have the following values:

  0, 1, 2, … , lim_i - 1

Recap from previous course

One-dimensional matrices (vectors)

- The vectors are one-dimensional arrays.

- Syntax:

type name_vect [lim];

Recap from previous course

The relationship between pointers and vectors

- The name of an array is a pointer because it holds the value the address of its first element. However there is a difference between an array name and a pointer type variable. Thus, a pointer type variables can be assigned values at runtime, while it is not possible to make the same for an array/vector (the name of a vector/array always represents the address of its first element).
- It is customary to say that the name of an array is a **constant pointer**.

Recap from previous course

How to use a vector as a parameter to a function

- as a pointer
- as a vector with unmentioned dimension
- as a vector with mentioned dimension

Recap from previous course

const modifier

- A variable can be transformed into a constant value (that cannot be modified).

- Usually a constant is defined using

  # define

- But **const** modifier can also transform a variable into a constant.

Recap from previous course

Strings in C

- The character string is one of the most widely used applications that involves vectors.

- A string in C is an array of char values terminated by a special null character value **'\0'**.

Example: a statically declared string that is initialized to "hi":

```
char str[3];

str[0] = 'h';

str[1] = 'i';

str[2] = '\0';

printf("\n %s \n", str);
```

Main properties of strings:

1. A string is stored in a memory organized as a vector of ***char*** type.
2. Each character is kept on different byte (through its successive code number).
3. The most commonly used code for this purpose is ASCII.
4. After the last character of the string there must be added the null character, which is '**\0**'.
5. Since there is null in final position, then we have to declare the character array with one more character than the effective number of characters.
6. Although C does not have explicit data string, it allows constant string. A string constant is a list of characters enclosed in quotation marks (eg "hello").
7. Often it is not necessary to be entered manually the null character at the end of the string (the compiler does automatically). This happens when using special functions to operate strings (like **gets()**).
8. To operate a string you can use:
   - The name of the array (the constant pointer to the string respectively);
   - A simple pointer to that string.

Example

If we declare:

char tab[ ] =”This is a string”;

then:
tab – is the address of first char `T`
tab+1 – is the address of second char `h`
tab+2 – is the address of third char `i`
…
etc.
or:
tab[0] → contains the ASCII of first char `T`
tab[1] → contains the ASCII of second char `h`
…
A similar effect we can obtain by using:

char const *p = ”This is a string”;

where
*p[0] or *p - contains the ASCII of first char `T`*
*p[1] or *(p+1) - contains the ASCII of second char `h`*
…

Recap from previous course

Standard functions for strings

- C provides a library for strings (**string.h**).

- C string library functions do not allocate space for the strings they manipulate, nor do they check that you pass in valid strings; it is up to your program to allocate space for the strings that the C string library will use.

- Calling ***string*** library functions with bad address values will cause a fault or "strange" memory access errors.

Some of the functions in the Standard C string library

- calculating the length of strings (eg strlen)
- copying the strings (eg strcpy)
- concatenating the strings (eg strcat)
- comparing the strings (eg strcmp)
- searching (identification) of character or substrings (eg strchr, strstr)

1. Computing the string length

- The length of a string is the number of the own characters which fall within its composition.
- NULL character is not considered in determining the length of a string.
- NULL's presence is necessary because in determining the length of a string are counted all characters until its occurrence (but without NULL).

Syntax:

unsigned strlen(const char * s);

Examples:

1) *char * const p = ”This is a string”;*
unsigned n;
...
n=strlen (p);
/* n will be assigned with the value 16 */

2) *char tab [ ] = ”This is a string”;*
int n;
n = strlen(tab);
/* n will be assigned with the value 16 */

3) *int n;*
n = strlen(”This is a string”);
/* n will be assigned with the value 16 */

Notes:

- All three examples had identical results.
- The **strlen** function formal parameter is a constant pointer.
- Consequently, **strlen** function must not modify the characters string which is of a determined length.

2. Copying the strings

- Sometimes it is necessary to copy a string of character from a memory area in another part. For this we use the strcpy function, which has the syntax:

char * strcpy (char * dest, const char* source);

- The function makes a copy of the string pointed by the **source** to the memory area whose address is indicated by **dest**.
- The function copies all the characters, including the null one, from the end of the string.
- The function returns the address of **dest**.

Examples

1)
```
char tab[ ] = “This string is copied”;
char t [ (sizeof tab)+1];
…
strcpy (t, tab);
```

2)
```
char t [100];
strcpy (t, “This string is copied”);
```

3)
```
char * p=“This string is copied”;
char t [100];
char * q;
q=strcpy(t, p);
```

Variants of copy function

- In order to copy only first n characters of a string from one area to another zone of memory, we can use the function:

char *strncpy (char *dest , const char * source , unsigned n);

- If *n* ≥ *string length* - then all characters of the string are transferred to the **dest**;
- If *n* < *length of the string* - then copy only the first *n* characters of the string.

Example 1

*char * p = “This string is partially copied”;*
char t[12];
strncpy (t, p, (sizeof t)-1);

Example 2

- Write a program that reads a sequence of words, and finally displays only the longest word.

```
#include<stdio.h>
#include<string.h>
#define MAX 100      /* it is assumed that a word has no more than
100 characters*/
int main( )
{
 int max=0, i;
 char word[MAX+1];
 char word _max[MAX+1];
 while (scanf("%100s", word)!=EOF)
          if (max < (i = strlen(word)))
                    { max= i;
                       strcpy (word _max, word);
                    }
 if (max)  printf("The longest word is %s and its length is %d \n",
                              word _max, max);
getch();
}
```

3. Concatenating the strings

- For concatenating strings we can use **strcat** function.

- Syntax:

char *strcat(char *dest, const char *source);

Notes:

- **strcat** function copies the string from the source area to memory zone, immediately following the last character of the string, which was already on dest.
- The function returns a pointer to dest.
- It is assumed that the area pointed by **dest** is large enough to keep all characters from these two strings which are concatenated plus the NULL character.
- **strcat** function does not modify the string pointed to by the source (which is a constant pointer).

Example

```
…
char tab1[100] = “The language C++”;
char tab2[ ] = “is a superset of C”;

        …

strcat(tab1, “ “);
strcat(tab1, tab2);
```

Variant of **strcat** function

char *strncat (char *dest, const char *source, unsigned n);

- If *n* ≥ *length of source* - then all characters of the source string are concatenated after **dest**;

- If *n* < *length of source* - then only the first *n* characters of the source string are concatenated after **dest**.

4. Comparing the strings

- Strings of characters can be compared using the ASCII characters in their structure.

- To better understand this mechanism we consider two strings: **s**1 and **s**2.

- We have the following cases:

two strings: **s**1 and **s**2

- **s1 = s2** – if both have similar length and s1[i] = s2[i] ∀ i.

- **s1 < s2** – if ∃ ***i*** such that *s1[i]<s2[i]* and *s1[j]=s2[j]* ∀ j= 0, 1, …, i-1.

- **s1 > s2** – if ∃ ***i*** such that *s1[i]>s2[i]* and *s1[j]=s2[j]* ∀ j= 0, 1, …, i-1.

We can think of the ordering of words in a dictionary…

strcmp function

int strcmp(const char *s1, const char *s2);

- This function returns:
  - a negative value - if s1 < s2
  - 0 - if s1 = s2
  - a positive value - if s1 > s2

Example

```
...
char *p=”Student”;
char name[20];
…
printf(“\n Password: “);
gets(name);
if (strcmp(name, p))
        exit(1);
…
```

stricmp function

int stricmp(const char *s1, const char *s2);

- This function is similar to ***strcmp***. The only difference is that ***stricmp*** function does not distinguish between uppercase and lowercase letters.

- *Note: there is **strcmpi** on some compilers*

Example

Considering:

…
char *sir1 = “ABC”;
char *sir2 = “abc”;
int i;
…

By using: ***i = strcmp(sir1, sir2);***
we obtain a negative value, since “ABC”<”abc” (***A*** has 41h, while ***a*** has 61h)

But when: ***i = stricmp(sir1, sir2);***
we obtain zero

strincmp function

int strincmp(const char *s1, const char *s2, unsigned n);

- This function limits the comparison up to ***n*** characters (and also ignores the difference between uppercase and lowercase letters).

- *Note: there is **strncmpi** or **strncmp** on some compilers*

Example

- Write a program that reads a sequence of words, and finally displays only the greatest word from that sequence.

```
#include<stdio.h>
#include<string.h>
#define MAX 100

int main ()
{ char word[MAX+1];
  char word _max[MAX+1];
 word _max [0]='\0'; /* word _max is initialized with null */

  while (scanf("%100s", word)!=EOF)
        { if (stricmp(word, word _max)>0)
              strcpy (word _max, word);
         }
   printf("\n The greatest word is  %s", word _max);

   getch();
   return 0;
}
```

5. Finding strings or characters

- Some useful functions:

strchr(s1, ch); /* return a pointer at first finding of ***ch*** character in ***s1*** */

strstr(s1, s2); /* return a pointer at first finding of ***s2*** inside ***s1*** */

- If there is no match (finding), then these functions return zero.

Syntax:

char *strchr(const char *s1, const char ch);

char *strstr(const char *s1, const char *s2);

Example

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

int main ()
{
 char s[80];
 gets(s);          /* write a text */
...
 if (strchr(s, 'e')) printf("The character - e - was find in the string");

 if (strstr(s, "test")) printf("The word - test -  was find in the string");
...
}
```

Initializing Arrays

- In C, an array can be initialized, either one by one element, or using a single statement as follows:

```
double A[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

- The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

Two-Dimensional Arrays

- The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size **lim_1**, **lim_2** you would write something as follows:

type *arrayName*[lim_1][lim_2]