Computers Programming

Course 10

Iulian Năstac

Recap from previous course

4. Calling through the arguments of a function

Calling a function in C can be realized (relative to the nature of his arguments) in two ways:

- by value;

- by reference.

a. Calling by value

If data is passed by value, the data is copied from the variable used in (for example main()) to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

b. Calling by reference

When the function uses a pointer parameter then in that formal parameter it is actually copied an address of the memory. The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

Recap from previous course

5. Values returned by a function

- A **return** statement causes execution to leave the current subroutine and resume at the point in the code immediately after where the subroutine was called, known as its return address.
- The return address is saved, usually on the process's call stack, as part of the operation of making the subroutine call.
- Return statements in C allow a function to specify a return value to be passed back to the code that called the function.

Recap from previous course

6. Recursive functions

- Recursion is a programming technique that allows the programmer to express operations in terms of themselves.
- In C, this takes the form of a function that calls itself.
- A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process".
- This makes it sound very similar to a loop because it repeats the same code, and in some ways it *is* similar to looping.

Recap from previous course

7. The efficiency of the functions

Functions are essential to write effective programs (unless very simple ones). However, in some special applications it is better to remove and replace a function with an **inline** code.

Inline code performs the same actions as the function without the disadvantage caused by a function call (including the decreasing of speed).

8. Variadic functions

- A variadic function is a function that accepts a variable number of arguments.
- A good example, among specific operations, that has been implemented as a variadic function in many languages is output formatting. The C function **printf** format is such an example.
- To portably implement variadic functions in the C programming language, the standard **stdarg.h** header file is used. The older **varargs.h** header is still in use for some compilers.

Recap from previous course

There are 4 storage classes in C:

- auto
- register
- static
- extern

Notes:

- Global variables (static and extern) are initialized to 0 automatically
- Local variables get arbitrary values, depending on actual bits' configuration on stack memory

Recap from previous course

Initialization

- Initialization is the assignment of an initial value for a data object or ***variable***.

- Syntax:

  class type name = expression;

Recap from previous course

Notes (again):

- Avoid to use local variables (auto) which are not initialized. These variables get arbitrary values when are initialized. This is due to the fact that on the stack when a variable is erased, this doesn't means that their position on the stack will become zero.
- Global and static variables always get zero value at declaration.
- **Function parameters cannot be initialized**.

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.

Notes:

- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

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

Notes:

- Each lim_i (where i = 0, 1, 2, …, n) is a positive integer.
- We can access the elements of a matrix by using variables with indices.
- In C, the name of a matrix is in fact a pointer that indicates the address of its first element.
- At the declaration of a matrix, the compiler allows a memory zone to contiguous store all its elements.
- **Matrices and pointers are strong related**.

Example

- **int a[3][4];**
  this declares an integer array of 3 rows and 4 columns. Index of row will start from 0 and will go up to 2.

	Column 0	Column 1	Column 2	Column 3
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

One-dimensional matrices (vectors)

- The vectors are one-dimensional arrays.

- Syntax:

  type name_vect [lim];

Notes:

- The declaration:

  ```
  char p[10];
  ```

  creates a matrix of 10 elements, from p[0] to p[9].

- The amount of memory required for recording a matrix is proportional to the type and its size. The total size in bytes is calculated:

  No_of_bytes = sizeof(type) * lim;

- C does not control the limits of the array. You can overcome both edges of a matrix and write in wrong place (of other variables). It remains the responsibility of the programmer to control limits, where necessary.

Example of a mistake:

```
…
int mat[100], i;
…
for (i=0; i<=100;  i++)
        mat[i]=i;

…
```

- A vector is actually a list of information of the same type stored in contiguous memory locations, in the order of indices.

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**.

Example:

```
……….
int t[10];
int *p;
int x;
……….
p=t;
……….
x=t[0];
```

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

Examples:

a) *void funct (int *x)* /* pointer*/

```
        {...........
        }
```

b) *void funct (int x[ ])* /* vector with unmentioned dimension*/

```
        {...........
        }
```

c) *void funct (int x[10])* /* vector with mentioned dimension */

```
        {..........
        }
```

Notes:

1) All three methods of declaration (previously exemplified) will provide similar results because each sends the compiler a pointer to an integer value, which is the first element of the vector (in fact, in the first example, a pointer is clearly used).
2) In case of use as a function argument, the size of the matrix does not really matter, because C does not control the boundaries. It will be absolutely correct to use, for instance, the form:

```
void funct (int x[50])
 {..........
 }
```

because C creates a code that instructs the function ***funct( )*** to receive a pointer to that array and does not actually create an array of 50 values.

<u>Example</u>:

Initial data:

- A program reads a vector of numbers and then displays them in ascending order.
- The maximum number of items is 1000.
- A function for ordering numbers will be used, which will be named:

```
void ordcresc(double tab[ ], int n)
```

```
#include <stdio.h>
#include<conio.h>
#define MAX 1000
void ordcresc(double tab[ ],int n);
int main()
{ double v[MAX];
  int m,s,i;
  printf("\ Enter the number of elements = ");   scanf("%d", &m);
  printf("\n Enter the vector elements: \n");
  for(s=0 ; s<m ; s++)
    scanf("%lf",&v[s]);
    ordcresc(v,m);
    for(i=0;i<m;i++)
    {  printf("v[%d]=%g\n",i,v[i]);
       if((i+1)%23==0)
               {  printf("\n Press a key to continue \n");  getch();
               }
    }
    getch();
}
…
```

```
…
void ordcresc(double tab[ ], int n)
 /* Sort the elements of tab in ascending order */
 {
   int i,ind;
   double t;
   ind=1;
   while(ind)
     {
       ind=0;
       for(i=0;i<n-1;i++)
              if(tab[i]>tab[i+1])
               { t=tab[i];
                 tab[i]=tab[i+1];
                 tab[i+1]=t;
                 ind=1;
               }   /*end if*/
     }  /*end while*/
 }
```

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.

Possible syntaxes:

1) type ***const*** name = value;

2) type ***const*** *name = value;

3) ***const*** type name = value;

4) ***const*** type *name = value;

5) ***const*** name = value;

Notes:

- Once established the variable (which is const), further assignments, like:

  **name='a';* or

  **(name+1)='b'*

are incorrect.

- If the assignment is missing, then it means that we are dealing with a formal parameter of a function:

  *tip f (const tip *name);*

Attention

- It is possible to circumvent the restrictions imposed by the modifier const. If you use another variable that refers to the same memory location, then that location (other pointer) may alter its content:

```
...
char const *s = ”string of char”;
char *p;
p = (char *)s;
*p = ’a’;
*(p+1) = ’b’;
...
```

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`*

…

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.h*** 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 (e.g. strlen)
- copying the strings (e.g. strcpy)
- concatenating the strings (e.g. strcat)
- comparing the strings (e.g. strcmp)
- searching (identification) of character or substrings (e.g. strchr, strstr)