French Azerbaijani University
Arash Habibi
Second year
2017-2018

Advanced C Programming

Practical Work 3 : pointers and dynamic allocation

Sections 1, 2 and 4 consist of implementing and testing the questions of exercice session 4.

1 Passing arguments by reference

1. Write a function with prototype `int divide(float a, float b, float *q);` which divides `a` by `b` and puts the result at the `q` address. The returned value is a diagnosis (it says whether everything went right or not). The function returns 0 if `b=0` (impossible division) and 1 otherwise.
2. Write an example of how we can use this function to divide 365 by 5 and to print the result.

2 Functions providing structures or arrays

Consider the following function where vectors are represented by an array :

```
float* V_new1(x,y,z)
{   float thenew[3];
    thenew[0] = x;
    thenew[1] = y;
    thenew[2] = z;
    return thenew;
}
```

And consider the following code where vectors are represented by a structure :

```
struct vector { float x,y,z; };

struct vector V_new2(x,y,z)
{   struct vector thenew;
    thenew.x = x;
    thenew.y = y;
    thenew.z = z;
    return thenew;
}
```

1. Try to answer this question without programming : what should be the output of the following call :

   ```
   float *v1 = V_new1(5,6,7);
   struct vector v2 = V_new2(1,2,3);
   printf("v1 : (%f,%f,%f)\n",v1[0],v1[1],v1[2]);
   printf("v2 : (%f,%f,%f)\n",v2.x, v2.y, v2.z);
   ```

2. In particular, is there any problem with `V_new1` and `V_new2` returning the address (`V_new1`) and the value (`V_new2`) of a local variable ?
3. Now try the program on your computer and see the output. Is the output in accordance with your expectations ?
4. How can you rewrite `V_new1` in order to avoid the problem.

3 Dynamic allocation

1. Dynamically allocate an array of 5 integers ;
2. Dynamically allocate an array of 7 chars ;

4 Stack

In the last practical work session, you dealt with the following structure :

```
struct stack
{    int _size;
     int _array[300];
};
```

300 is the maximum number of elements that it is possible to store in the stack. We wish to modify this structure in order to be able to choose the maximum size of the stack. In other words, we don't want the maximum size to be always 300. We will need to take smaller or bigger stacks depending on user input. Here is the new structure :

```
struct stack
{    int _size;
     int *_array;
};
```

We also change the prototype of this function : `struct stack ST_new(int maxsize);`. Rewrite the `ST_new` function so that the `_array` field of the stack has `maxsize` elements.

5 Dynamic allocation of 2D arrays

We wish to dynamically allocate and manipulate a 2D array of floats. Write a program which :

1. starts by asking the user (`scanf` function) the value of `n` (width of the array) and `m` (height of the array) ;
2. dynamically allocates a 2D array of `n` columns and `m` rows of floats (`n` and `m` are integer variables). Choose different values for `n` and `m`. *Hint :* maybe it will be easier to define the array of pointers first.
3. Write a function that achieves this allocation. What should be the prototype of this function ?
4. Write a function with prototype `void printIt(float **grid, int n, int m);` which prints the contents of the 2D array on the terminal.
5. Write a function with prototype `void fillIn(float **grid, int n, int m);` which fills the array with any specific value (for example let the element of index `i,j` have a value of `i+j`).

6 Program arguments

6.1 `argc` and `argv`

In this section, we will see how we can use the arguments of our programs. Consider the following program :

```
#include <stdio.h>
int main(int argc, char **argv)
{    printf("argc=%d\n",argc);
     for(int i=0; i<argc; i++)
          printf("%s\n",argv[i]);
     return 0;
}
```

This is probably the first time that you use the `main` function with arguments. Copy this code in a file and compile it. Let `calc` be the name of the executable file. Try typing :

```
./calc
./calc toto titi blabla
./calc ufaz 3 + 2534 = 9
```

Now you probably understand the meaning of `argc` and `argv`. If you don't, then ask the question from your teacher or from internet.

6.2 Functions `atoi` and `atof`

This section contains no questions. Simply an introduction to these two functions. They are declared in `stdlib.h`. So you will need to include that file if you use those functions.

— `atoi` means *Ascii To Integer* and converts a character string representing an integer to an integer ; For example `atoi("423")` is 423 (an integer) ;

— `atof` means *Ascii To Float* and converts a character string representing a floating point number to a float ; For example `atoi("3.5")` is 3.5 (a float).

6.3 Calculator

We wish to program a simple calculator, capable of achieving the four main operations $(+, -, \times, \div)$ on two numbers. But we do NOT want to use `scanf`. We want the program to use its arguments. For example if the name of our executable file is `calc`,

— The output of `./calc 3 + 2.5` should be `5.500` ;

— The output of `./calc 3 - 2.5` should be `0.500` ;

— The output of `./calc 3 x 2.5` should be `7.500` ;

— The output of `./calc 3 / 2.5` should be `1.200` ;

Note that the operands and the operators must always be separated by a space. Note also that the user will not represent the multiplications by the '*' character but by the 'x' character. This is because '*' has a specific meaning in `bash`. But in C language of course, the multiplication operator remains '*'.

1. Which element of `argv` represents the operator ?
2. Extract the first character of that element and identify the operator ;
3. Which elements of `argv` represent the first and second operand ?
4. Convert the operands to floats ;
5. Carry out the operation and print the result.