French Azerbaijani University
Arash Habibi
Second year
2017-2018

Advanced C Programming

Exercice session 4 : Memory, addresses and pointers

Before you start this session, make sure you have read chapter 5.

1 Operators & and *

1. — Declare an `int` variable `n` ;
   — Declare another variable `pn` that would be a pointer to an int ;
   — Make `pn` point at `n` (in other words, let the value of `pn` be the address of `n`).

	addresses	data
	...	...
b	1001	12
a	1000	35
	...	...
p1	...	...
p2	...	...
	...	...

You may find the first two questions of this section very easy. But the following ones will be more difficult.
I strongly recommend that you draw the state of the memory at each instruction. Let us suppose that the address of `a` is 1001 and the address of `b` is 1000 (or any other value). Put also `p1` and `p2` somewhere in the memory.
At each instruction write what variables are changed, in order to find the correct answer.

What would be the output of each of the following instructions ?

2.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
printf("%d %d\n",*p1, *p2);
```

3.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
p1 = p2;
printf("%d %d\n",*p1, *p2);
```

4.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
p1 = p2;
p2 = &a;
printf("%d %d\n",*p1, *p2);
```

5.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
*p1 = 10;
printf("%d %d\n",*p1, *p2);
```

6.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
p1 = p2;
*p1 = 10;
printf("%d %d\n",*p1, *p2);
```

7.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
*p1 = *p2+10;
*p2 = 5;
printf("%d %d\n",*p1, *p2);
```

8.
```
char a=35, b=12;
char *p1, *p2;
p1 = &a;
p2 = &b;
*p1 = *p2+10;
*p2 = *p1*2;
printf("%d %d\n",*p1, *p2);
```

9.
```
void func(int var)
{    var = 5;   }

int main()
{   int var=0;
    func(var);
    printf("var=%d\n",var);
    return 0;
}
```

10.
```
void func(int *var)
{    *var = 5;   }

int main()
{   int var=0;
    func(&var);
    printf("var=%d\n",var);
    return 0;
}
```

11.
```
void func(int *foo)
{    foo[0] = 50;   }

int main()
{   int array[10];
    array[0] = 0;
    func(array);
    printf("array[0]=%d\n",array[0]);
    return 0;
}
```

In the following questions, you must say whether the output is `NULL` (or `0x0`) or a valid address.

12.
```
void func(int *foo)
{    foo = (int*)calloc(5,sizeof(int));  }

int main()
{   int *array = NULL;
    func(array);
    printf("array=%p\n",array);
    return 0;
}
```

13.
```
int* void func()
{    int *res = (int*)calloc(5,sizeof(int));
     return(res);
}

int main()
{   int *array=NULL;
    array=func();
    printf("array=%p\n",array);
    return 0;
}
```

14.
```
void func(int **foo)
{    *foo = (int*)calloc(5,sizeof(int));  }

int main()
{   int *array = NULL;
    func(&array);
    printf("array=%p\n",array);
    return 0;
}
```

2 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.
3. Write a function with prototype : `float firstDegreeEquation(float a, float b);` which, for two given numbers `a` and `b`, solves the following equation :

$$ax + b = 0 \tag{1}$$

   and returns the solution. For example `firstDegreeEquation(1,5)` should return -5. What do you think we could do when there is no solution (when `a` is 0) ?
4. The problem with the previous function is that, when `a=0` and `b!=0`, there is no solution, and when `a==b==0`, any value of `x` is a solution. So, even if we print an error message, the function has to return some value : 0 ? -1 ? And these values could be interpreted by the calling function as valid solutions. This is why we write the function in another way. Write a function with prototype :

```
int firstDegreeEquation(float a, float b, float *solution);
```

which returns 0 if the equation has no solution, 1 if it has one solution and 2 if any number is a solution. When the solution is valid, it is put at the address given by `solution`.

5. Write an example of the use of such a function.

3 Dynamic allocation

1. Dynamically allocate an array of 5 integers ;
2. Dynamically allocate an array of 7 chars ;
3. The `Color` structure is defined as follows :

```
struct Color
{   unsigned char r,g,b; };
Dynamically allocate an array of 10 \verb+Colors+ ;
```

4. Write a function with prototype `int* newIntArray(int n);` which returns a newly allocated array of `n` integers.

4 Dynamic allocation of 2D arrays

Now we need to allocate a 2D array. This means that the stored data is identified by two indices. For example if the stored data is a the height field of a terrain (as in figure 1) each height value is indexed by `x` and `y`. Same thing for a rectangular *maze*. Each cell in the maze is characterized by a column `x` and a row `y`.
As for 1D (ordinary) arrays, if we already know the size of the 2D array at programming time, we can make a static allocation. For example the following declaration :

```
float height_field[100][150];
```

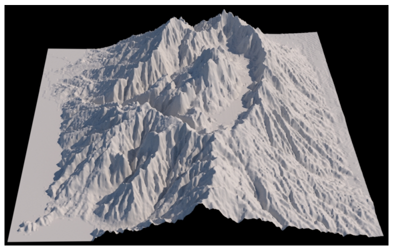
FIGURE 1 – A height field : to each value of `x` and `y` is associated a height value.

allocates a grid of 100×150 float numbers. Each number can be written as `height_field[x][y]` where `x` must be in the [0,99] interval and `y` in the [0,149] interval. But this does not work if we do not know the size of the array while we are programming (for example if the size depends on user input). In this cas, we need to allocate the array dynamically. For example let us take the height field example again. Instead of having an array of 100×150 floats, we would like to allocate an array of $N \times M$ floats.

`calloc` allocates 1D arrays. There is no equivalent of `calloc` for 2D arrays. We have to do it by ourselves. In other words, how can we make 2D arrays with a function that can only make 1D arrays ?

The idea is to make N 1D arrays. Each of these arrays would contain M floats (figure 4.a). We can do that by N calls to `calloc`. Let's suppose that `calloc` has produced N arrays : `arr1`, `arr2`, ..., in other words N pointers to float.

But building N individual arrays is not very practical. For example if we need the 2D array to be the argument of a function, the function would need to take N arguments. In order to have *one* data structure that would enable us to access the whole 2D array, we put our arrays `arr1`, `arr2`, ... in another array, that we call `height_field` (or `grid` as in figure 4.b). This is not an array of `float` any more, but an array of pointers to float (`float*`). What is the type of `height_field` ?

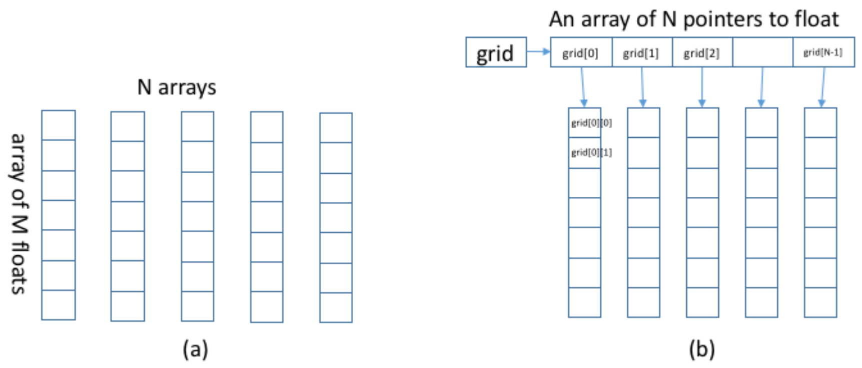

FIGURE 2 – In order to make a $N \times M$ array of floats, we make N arrays of M floats (a). And in order to access each array, we put the pointers of each array in an array of float pointers (b).

Thus `height_field[0]` represents the first array, `height_field[1]` represents the second array etc. Therefore since `height_field[1]` is an array, `height_field[1][0]` is the first number in this array. More generally, `height_field[n][m]` is the number index `m` in array index `n` or, in other words, the number on column `n` and row `m`.

1. Write a program which 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.
2. Write a function that achieves this allocation. What should be the prototype of this function ?
3. Write a function with prototype `void printIt(float **grid, int n, int m);` which prints the contents of the 2D array on the terminal.
4. 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`).

5 Pointers and arrays

1. Consider the following array which contains 5 elements.

   ```
   int my_array[5];
   printf("my_array=%p\n",my_array);
   ```

   it outputs : `my_array=0x7fff569bea40`. Could you give a pointer to the array containing the last 4 elements of `my_array` ? How about the last 3 elements of `my_array` ?
2. Write a recursive function with prototype :

   ```
   int belongs(int searched, int my_array[], int nb_elements);
   ```

   which returns 1 if integer `searched` is one of the elements of `my_array` and 0 in all other cases. Here is the recursive algorithm :

   (a) if `nb_elements=0`, then the array is empty, and the function returns 0 ;

   (b) otherwise (i.e. if there is at least one element in the array) if the first element is equal to `searched`, then the function returns 1 ;

   (c) otherwise, we call recursively this function with the following arguments :
   — `searched`;
   — an array containing the `nb_elements-1` last elements of `my_array` ;
   — `nb_elements-1`.