
# Pointers in C Language

Concha Batanero Ochaíta *(concha.batanero@uah.es)*

Helsinki Metropolia University of Applied Sciences

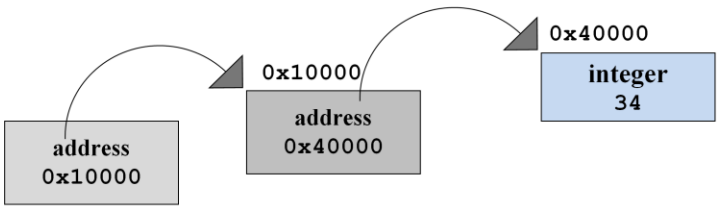
## Outline

- Concept of pointer to pointer
- Array of pointers
- Dynamic memory allocation. Two-dimensional arrays

## Concept

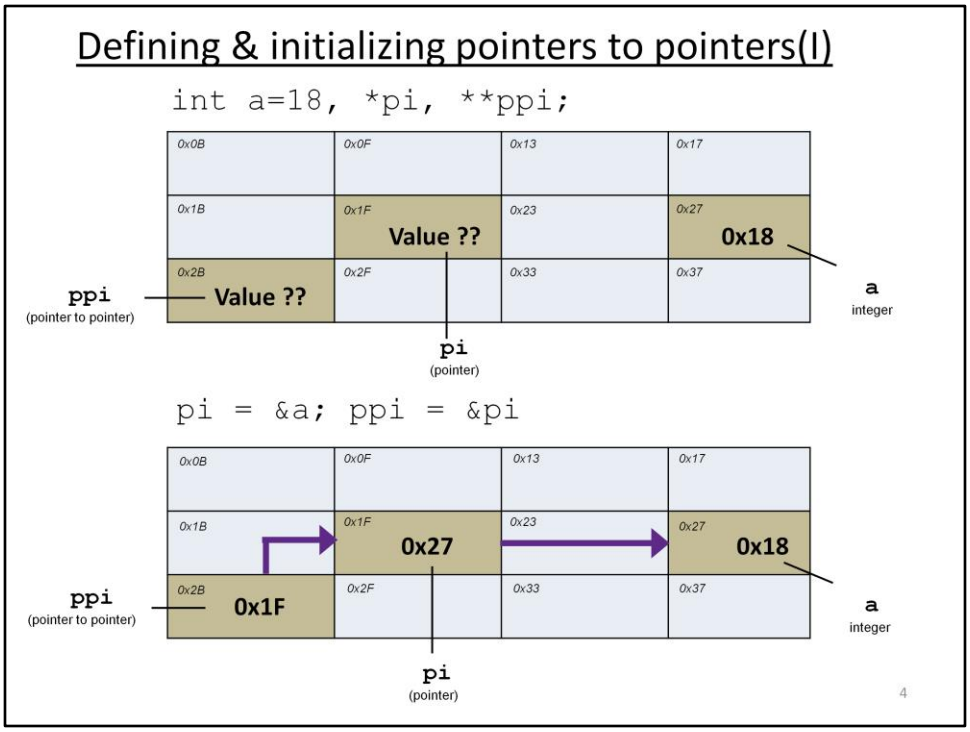
A pointer to pointer is a variable that stores a memory address where, in turn, is stored another address memory. A datum is stored on the second memory address.

```
int **pp;
```






A pointer to pointer is a pointer which points to another pointer that, in turn, is pointing to a datum.

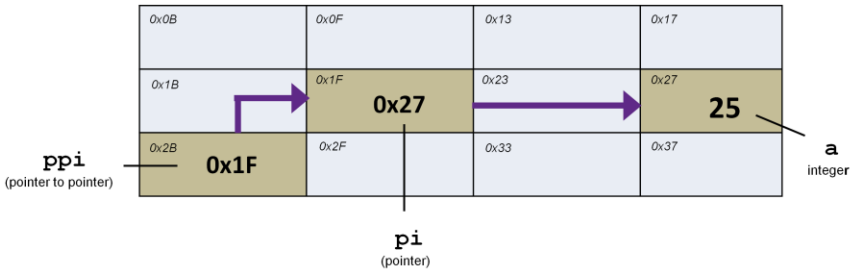



This slide shows the memory maps of different stages of three variables: integer, pointer to integer and pointer to pointer to integer. In the first stage the variables have been defined, but only the variable *a* has been initialized. In the second stage the pointers are initialized. Obviously the pointer to *int* (*pi*) must point to the *int* variable (*a*). The pointer to pointer to *int* (*ppi*) must point to the pointer to *int* (*pi*).

In the first image we don't know where the pointers are pointing because still a valid address has not been assigned to them. The second image shows the pointers pointing to a correct address through the arrows.

## Defining & initializing pointers to pointers (II)

```
**ppi = 25
```





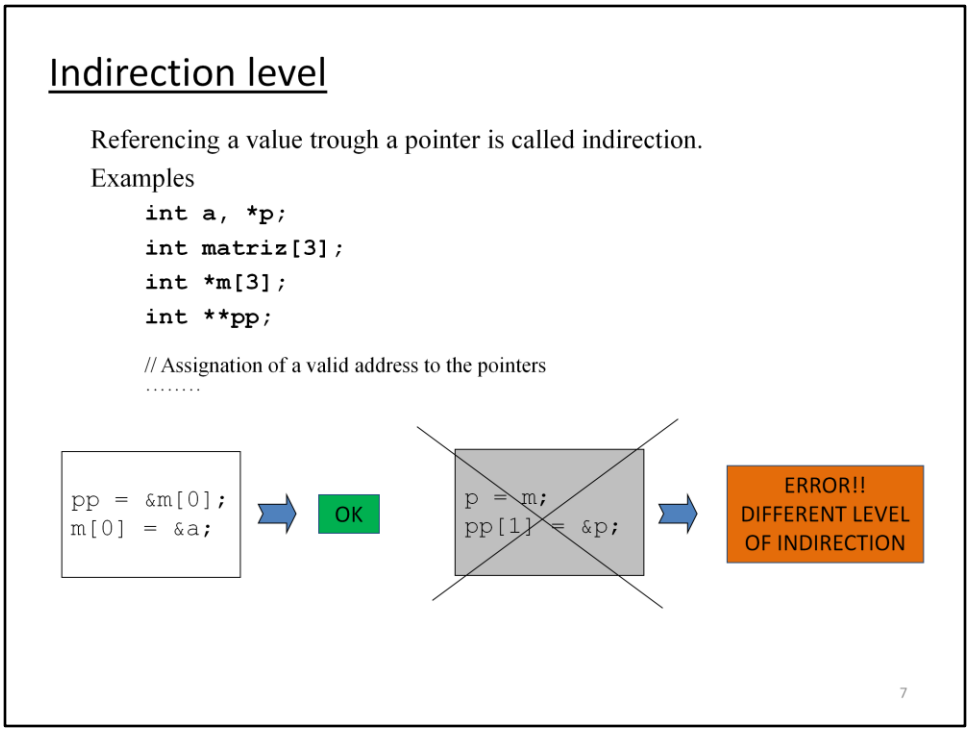

This memory map shows how has been affected the variable *a* when we have assigned the value *25* to the content of the content of *ppi*. That is so, because the content of *ppi* (**ppi*) is *pi*, also the content of *pi* (**pi*) is the variable *a*. Therefore, the content of the content of *ppi* (***pi*) is the variable *a*.

## An example

```
int a, *pi, **ppi;

// Initializing pointers
pi = &a;
ppi = &pi;

printf("Please, enter a number\n");

// any of the three scanf() statements carry out the same task
//scanf("%d",&a);
//scanf("%d",pi);
scanf("%d", *ppi);

// any of the three printf() statements print the same datum
 printf("\nThe entered number is: %d \n",**ppi);
 printf("\nThe entered number is : %d \n",*pi);
 printf("\nThe entered number is : %d \n", a);
```



Here there is an example which demonstrates that we said before.

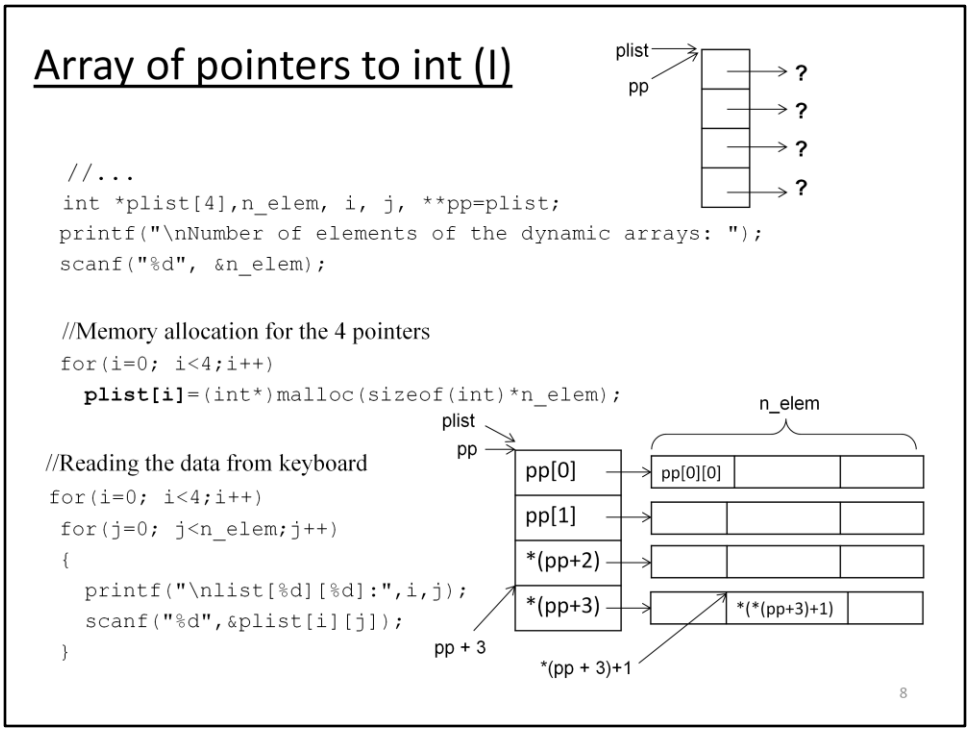



We can talk of indirection levels of the variables. Thus:

- A datum has 0 indirection level.
- A pointer has 1 indirection level.
- A pointer to pointer has 2 indirection levels.

The two expressions or operands of an equality must have the same level of indirection. That is, a datum only can be assigned to a datum, a pointer only can be assigned to a pointer, and a pointer to pointer only can be assigned to a pointer to pointer.

The slide presents some right and wrong examples.




This slide contains an example of a static array of pointers named *plist* which is composed by *4* pointers. Although the array is static, each of the *4* pointers will be a dynamic array since we are going to allocate memory for them. The first graphic, at the top of the slide, shows the static array when the dynamic memory has not been allocated yet. The graphic, at the bottom of the slide, shows the static array of pointers where each pointer points to the dynamic memory already allocated.

*plist* is a pointer to pointer since is the name of an array which elements are pointers (Remember: the name of an array is the pointer to the first element) . Therefore, the assignation of *plist* to the *pp* pointer to pointer is correct.

Once the arrays are allocated, the data are read from keyboard using the arrays notation. We could also use *pp* instead of *plist* since both are pointing to the array.

```
scanf("%d",&pp[i][j]);
```

The image shows examples inside its memory cells of how to access the elements in both cases the array notation and the pointers notation. The statement for reading the data with the pointers notation would be:

```
scanf("%d",*(pp+i)+j);
```

We focus on the element of the fourth row, second column to understand the last statement. *pp* points to the array. We can add *3* to *pp* and the pointer will point to the last element of the static array. If we apply the content to it, we will obtain **(pp+3)*, which is the last element of the static array and it is a pointer to *int*, since it is an array of pointers. If we add *1* to this pointer, it will point to the second element of the last dynamic array. To obtain de data that is inside of this memory cell we take the content:

```
*(*(pp+3)+1)
```

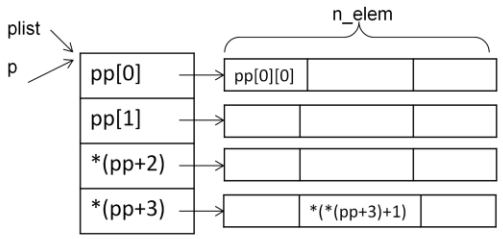
We have applied this concept in the case of the *scanf* statement, but in a general way, by using the index *i* to go along the rows and the index *j* to go along the columns. Moreover, in the *scanf* statement we must put the address of the datum. So, we can remove the asterisk at the left side and will obtain the address.

# Array of pointers to int (II)

```
//Data display
 printf("\n\n The data are:\n\n");
 for(i=0; i<4;i++)
 {
   for(j=0; j<n_eltos;j++)
     printf("%d\t",plist[i][j]);
   printf("\n\n");
  }

//Freeing memory
   for(i=0; i<4;i++)
     free(plis[i]);
 //...
```

Also possible:

```
printf("%d\t", pp[i][j]);

printf("%d\t",*(*(pp+i)+j));
```

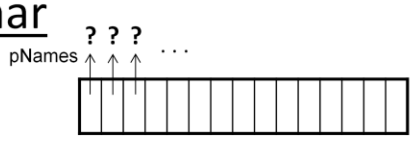


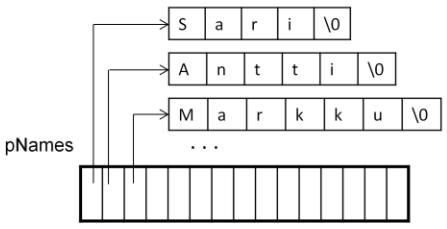

9

This slide continues with the example by showing the data and freeing memory. We have freed memory for pointers to which we had previously allocated memory for them. Notice that a static array no need to be freed.

# Array of pointers to char

```
char *pNames[15], name[20];

// read number of names (from 0 to 15)
for (i = 0; i < n; i++)
{
  printf("\nPlease enter a name: \n");
  fflush(stdin);
  fgets(name,20,stdin);
  pNames[i] = (char *)malloc(strlen(name) + 1);
  if (pNames[i] == NULL)
  {
    printf("Error allocating memory\n");
    for (j = 0; j < i; j++)
      free(pNames[j]);
    system("pause");
    return -1;
  }
  strcpy(pNames[i], name);
}
// print the list of names
// ....
```




10

This other example is about a static array of pointer to char. A pointer to char once is initialized is a string. The characteristic of the strings in this case, is that they are dynamic arrays and their size can be adjusted to the exact number of characters plus the character end of string (*0*) that the user introduces.

How can we know, as programmers, the number of characters of a string before the user has introduce it? In order to solve this problem, we always read from keyboard a static array named for example *name*. Then we allocated memory for the *strlen() + 1* of the *name* string and copy the *name* string into the allocated memory. Now we are ready to read other name from keyboard in the same string (*name*).

## Uses of a pointer to pointer (I)

1. To pass a pointer by reference to a function

```
int AllocateMemory(int **ppi);
```

2. To manage dynamic memory. Two-dimensional dynamic array

11

**1. Two different aproaches to allocate memory through a function**

We can use two different prototypes for a function that allocates memory. The function must return both the pointer which points to the allocated memory and the number of elements of the dynamic array. We can achieve this through two different prototypes of functions:

```
1.    int * AllocateMemory(int *pn);

2.    int AllocateMemory(int **ppi);
```

The first prototype returns the pointer pointing to the dynamic array and receives the number of elements by reference.

The second prototype returns the number of elements and receives by reference the pointer which will point to the dynamic array. Therefore, it receives a pointer to pointer. We must work inside the function with the content of the pointer to pointer if we want to access the pointer to the dynamic array of int. Here is the code of both functions:

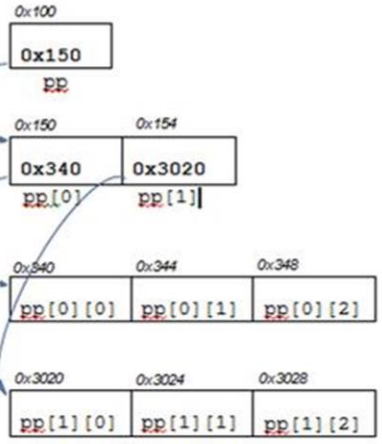
```
int * AllocateMemory (int *pnum)
{
   int *pi;
   printf("\nPlease, enter the number of elements: \n") ;
   scanf("%d", pnum) ;
   pi = (int *) malloc((*pnum)*sizeof(int)) ;
   if(pi == NULL)
   {
      printf("\nError allocating memory\n") ;
      *pnum = 0;
      return NULL ;
   }
   return pi;
}

int AllocateMemory (int **ppi)
{
   int num;
   printf("\nPlease, enter the number of elements:\n ") ;
   scanf("%d", &num) ;
   *ppi = (int *)malloc(num*sizeof(int)) ;
   if(*ppi == NULL)
   {
      printf("\nError allocating memory\n") ;
      return 0 ;
   }
   return num;
}
```

**2. To manage dynamic memory**

Next slides show schemes and examples of a two-dimensional dynamic array.

## dynamic two-dimensional array

```
int **pp;
```



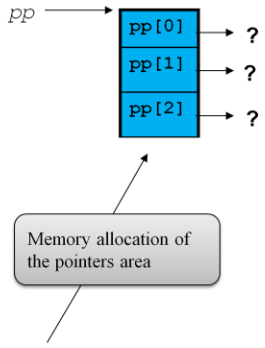

12

Here we have the memory map of a two-dimensional dynamic array of integers, as it is stored in memory. Slides number 8 and 9 show the rows in vertical position to a better understanding of the concept. But the real location in memory is how this picture is showing. Nevertheless the concept is the same. In this case, both the rows and the columns are dynamic. This example shows a two-dimensional array of 2*3 , that is, 2 rows and three columns. The array is pointed by the pointer to pointer *pp*;

*pp* points to a dynamic array of pointers to *int* which has two elements. In turn, each of the two pointers to *int* points to a dynamic array of three integer elements.

## Two-dimensional dynamic array (I)

```
int main()
{
  int **pp, i, j, n_rows, n_cols, k;

  // Reading of the number of rows and columns of the array
  printf("Enter number of rows\n");
  scanf("%d", &n_rows);
  printf("Enter number of columns\n");
  scanf("%d", &n_cols);

  //Memory allocation for the rows( elements of type int *)
  if ((pp = (int **)malloc(n_rows*sizeof(int *))) == NULL)
  {
    printf("Error allocating memory\n");
    return -1;
  }
```





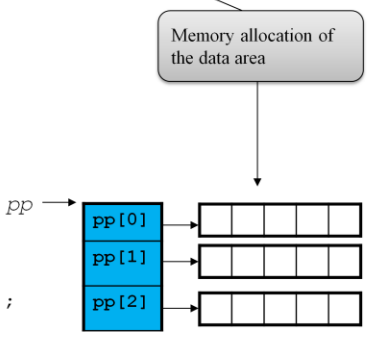

The next steps should be followed to create a dynamic two-dimensional array:

1. Define a pointer to pointer to the type of elements you wish to create the array.
2. Read from the keyboard the number of rows and columns of the dynamic array.
3. Allocate memory to the pointer to pointer. The number of elements, which are pointers, is the number of rows.
4. Allocate memory to each pointer. The number of elements is the number of columns.
5. Read the data of the two-dimensional array from the keyboard.
6. Display and/or operate with the data.
7. Free the memory.

This slide shows the three first steps. Notice that we should put *sizeof(int *)* in order *malloc()* allocates memory for pointers. Moreover the *void **  pointer returned by *malloc()* must be turned into pointer to pointer to int (*int ***)*

# Two-dimensional dynamic array (II)

```
// Memory allocation for the columns of every row
for (i = 0; i<n_rows; i++)
{
  if ((pp[i] = (int *)malloc(n_cols*sizeof(int))) == NULL)
  {
    printf("Error allocating memory\n");
    for (k = 0; k <= i - 1; k++)
      free(pp[k]);
    free(pp);
    return-1;
  }
}

// Reading of the data
for (i = 0; i<n_rows; i++)
  for (j = 0; j<n_cols; j++)
  {
    printf("pp[%d][%d]: \n", i, j);
    scanf("%d", &pp[i][j]);
  }
```






This slide shows the memory allocation for the pointers, as well as the reading of the data from keyboard. Notice that if the memory allocation for the pointers fail, we should free the memory previously allocated before finish the program. For example, if the allocation of the memory fails in *pp[2]*, we should free *pp[0]*, *pp[1]* and *pp*.

## Two-dimensional dynamic array (III)

```
  // Data display
   for (i = 0; i<n_rows; i++)
   {
     for (j = 0; j<n_cols; j++)
       printf("%3d", pp[i][j]);
     printf("\n");
   }

  // Freeing memory
   for (i = 0; i<n_rows; i++)
     free(pp[i]);
   free(pp);
   MemoryManager_DumpMemoryLeaks();
   return 0;
}
```



Finally, the data are displayed and the memory freed.