MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Pointers in C Language

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

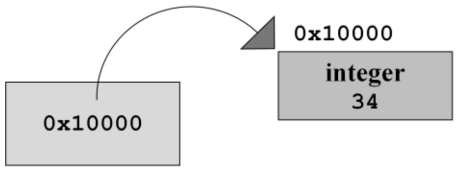
Helsinki Metropolia University of Applied Sciences

Outline

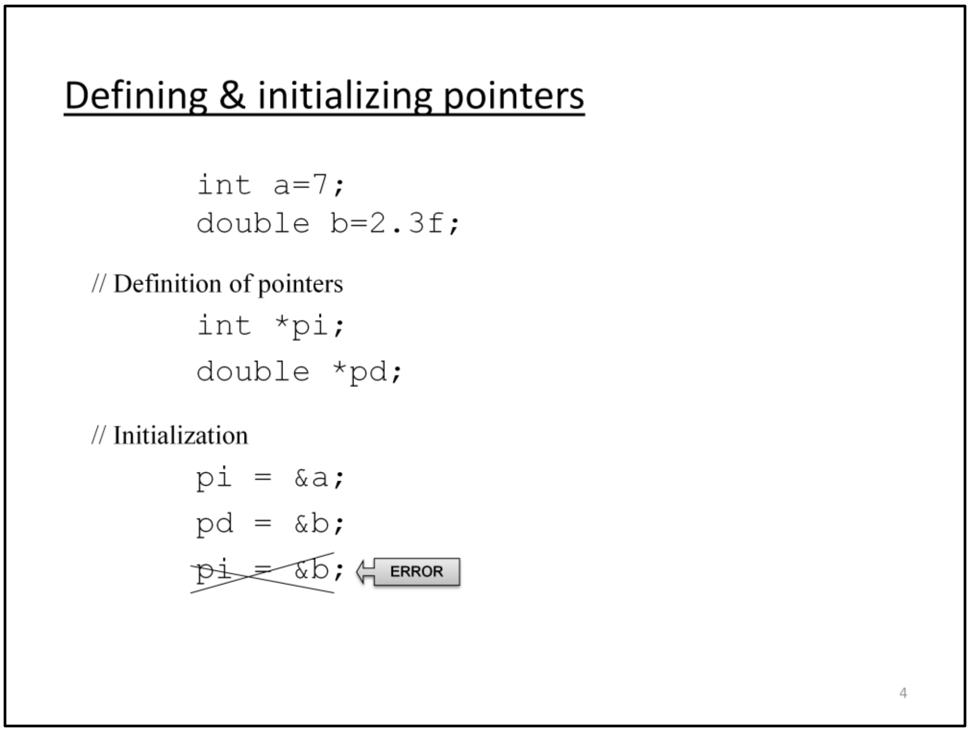
- General concepts of pointers
- Dynamic memory allocation
- Dynamic array of structures

Concept

A pointer is a variable that stores a memory address where, in turn, is stored a datum. It is said that the pointer points to this datum.



As it is said in the slide, a pointer is a variable, which value is a memory address. It means that the datum that is in this physical memory address is pointed by the pointer. A pointer is represented by an arrow.



Defining & initializing pointers

```
int a=7;
double b=2.3f;
```

// Definition of pointers

```
int *pi;
double *pd;
```

// Initialization

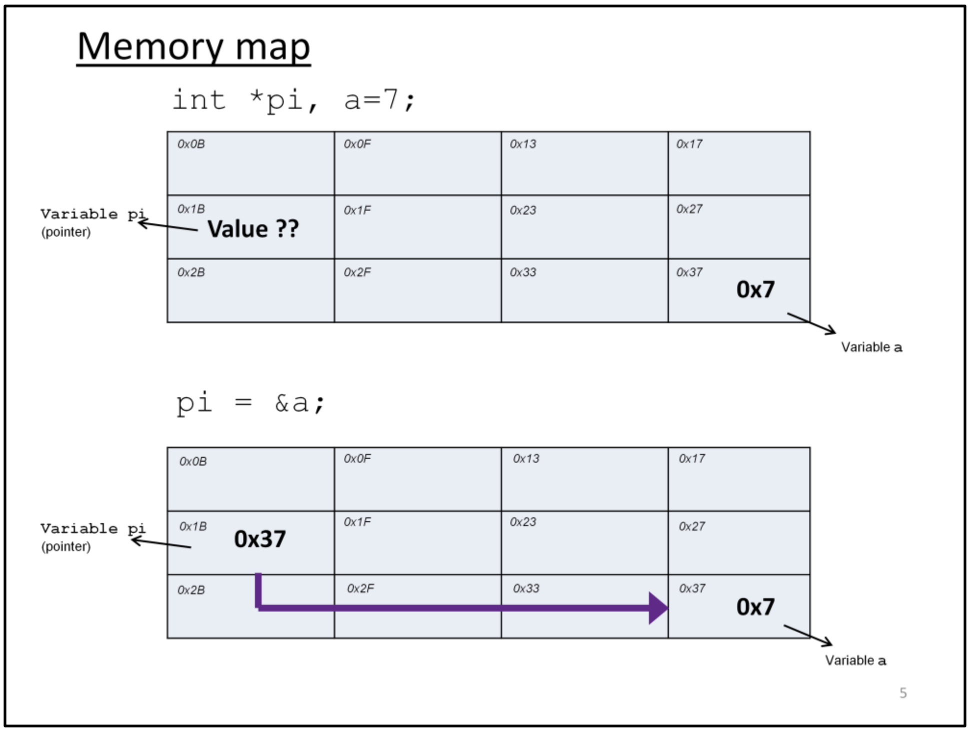
```
pi = &a;
pd = &b;
```

~~pi = &b;~~ ERROR

4

Here we have two variables of types *int* and *double*. They are named *a* and *b*. If we want to have two pointers pointing to each variable, we should do: firstly to define the two pointers and secondly to put these pointers pointing to the two variables. This second step is the initialization of the pointers and consists in giving to the pointers a valid address.

A pointer is defined in a special way: we must write the type of the variable that will be pointed by the pointer, then an asterisk and then the name of the pointer. In our example the pointer *pi* points to the variable *a* and the pointer *pd* points to the variable *b*. *pi* cannot point to *b* because *pi* must point to an *int* type and *b* is a *double* type.



Memory map

```
int *pi, a=7;
```

```
pi = &a;
```

We can see how the example works on a memory map.

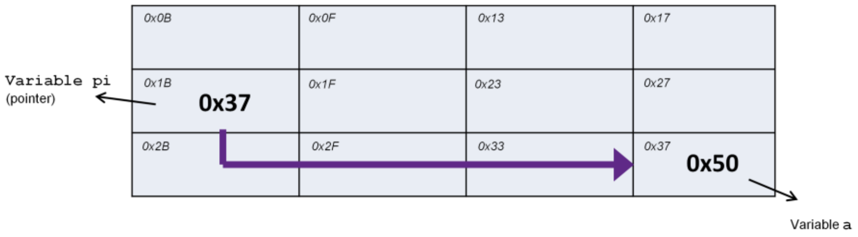
When we define variables, the operating system situates them in any available memory space. We cannot decide to assign a specific memory address, the operating system decides. For this example I have chosen on the memory map the address *37* for the variable *a* and the address *1B* for the pointer *pi*.

Once the variables have been defined we can see in the first memory map that the value of the variable *a* is *7* since it has been initialized to this value. But we don't know the value of the pointer because it hasn't been initialized. Therefore we don't know yet where the pointer is pointing and in consequence we cannot use it yet until it is initialized.

Now we give to the pointer a valid address (the address of the variable *a*) the pointer, will point to *a*. It can be seen on the second memory map.

Memory map

```
*pi=50;
```

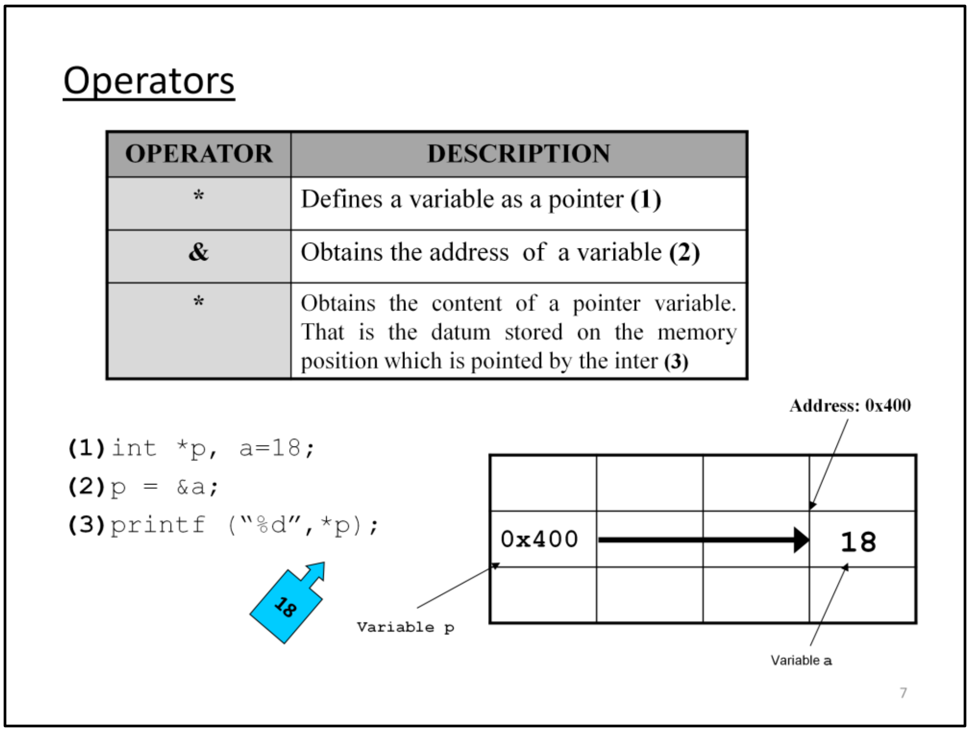


```
printf("%d", a);
printf("%d", *pi);
```

$\longrightarrow$ The two *printf* statements print the value 50 because both refer the same variable.

6

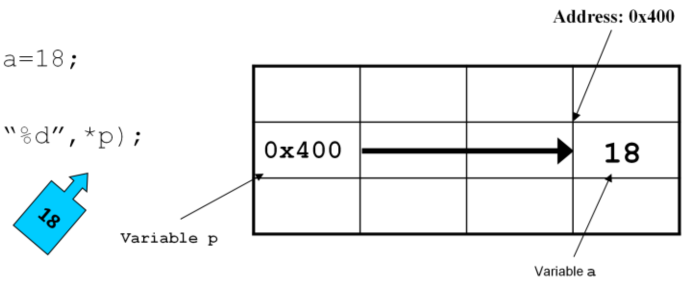
We continue with the example. Once the pointer has been initialized we can use it. We apply the asterisk operator to the pointer. **pi* is named *content of pi*. The content of pi (**pi*) is the value of the variable that is pointed by the pointer, i.e. the value of *a*. The memory map shows the result of the execution of the instruction:
**pi = 50;*



Operators

OPERATOR	DESCRIPTION
*	Defines a variable as a pointer **(1)**
&	Obtains the address of a variable **(2)**
*	Obtains the content of a pointer variable. That is the datum stored on the memory position which is pointed by the inter **(3)**

```
(1)int *p, a=18;
(2)p = &a;
(3)printf ("%d",*p);
```



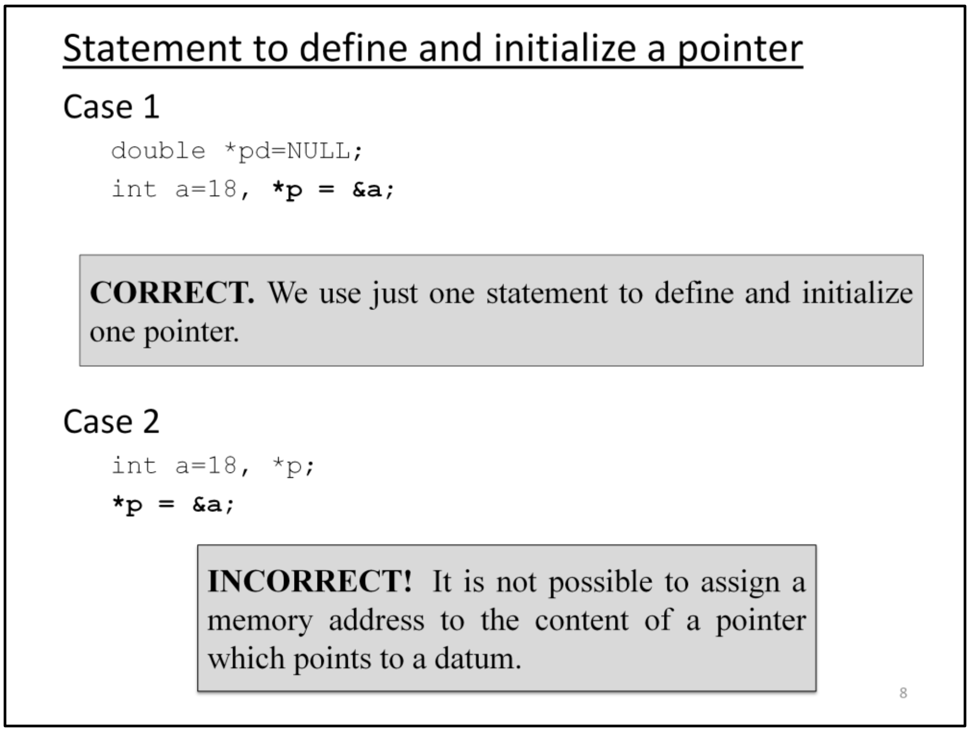
7

The table shows the operators we use when working with pointers. They are two: the asterisk and the ampersand.

The ampersand obtains the memory address of a variable.

The asterisk has two uses:

- One is to define one pointer.
- The other is to obtain the *content of the pointer*, that is, the value which is stored in the memory address pointed by pointer (shown on the last slide).



Statement to define and initialize a pointer

Case 1

```
double *pd=NULL;
int a=18, *p = &a;
```

CORRECT. We use just one statement to define and initialize one pointer.

Case 2

```
int a=18, *p;
*p = &a;
```

INCORRECT! It is not possible to assign a memory address to the content of a pointer which points to a datum.

8

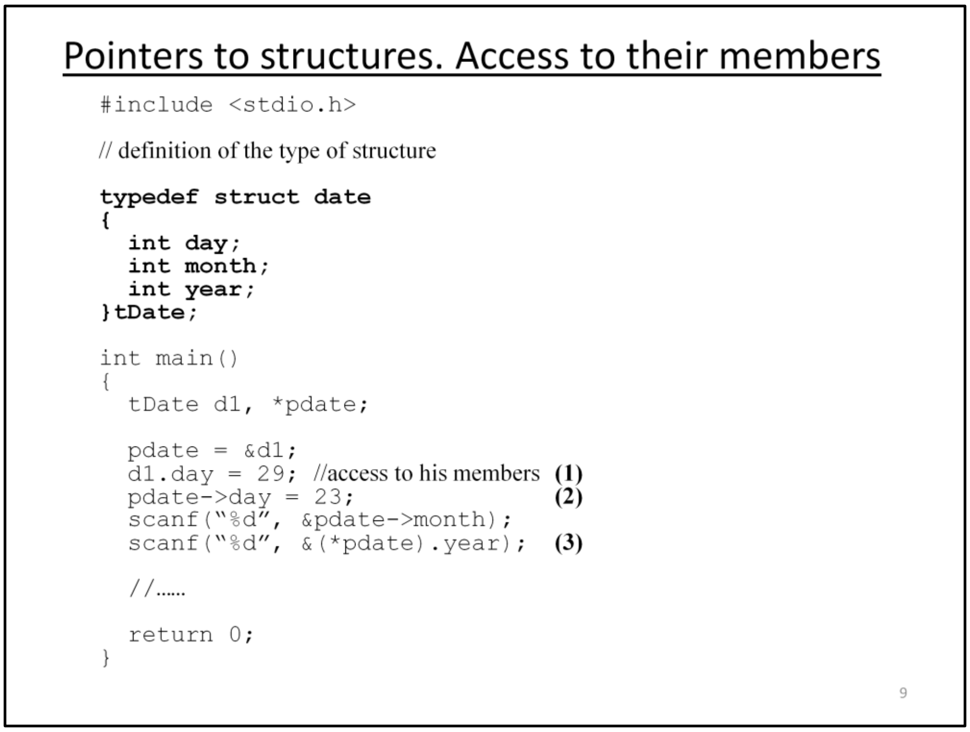
In the first case, we define and initialize two pointers. The pointer *pd* takes the initial value *NULL,* which means that the pointer does not point to any valid address. If we want to use the pointer *pd* we will have to assign it a valid address. The pointer *p* is defined and initialized to a valid address in the same statement. It is the same that we did with the variable a, which is correct.

In the second case there is not definition of pointer on the second statement. The pointer was defined in the line before. That means that we are trying to assign a memory address to the content of a pointer, which points to an integer. That is, we are trying to assign a memory address to an integer variable. It is wrong.

NOTE: For now a valid address is the address of one of the variables of the program. We cannot assign a physical memory address directly to a pointer such as this statement:

*int *pi = 0x10000;*

The operating system will never allow us to do that because this address could be busy with other program and we don’t know it. Only the operating system knows it.



Pointers to structures. Access to their members

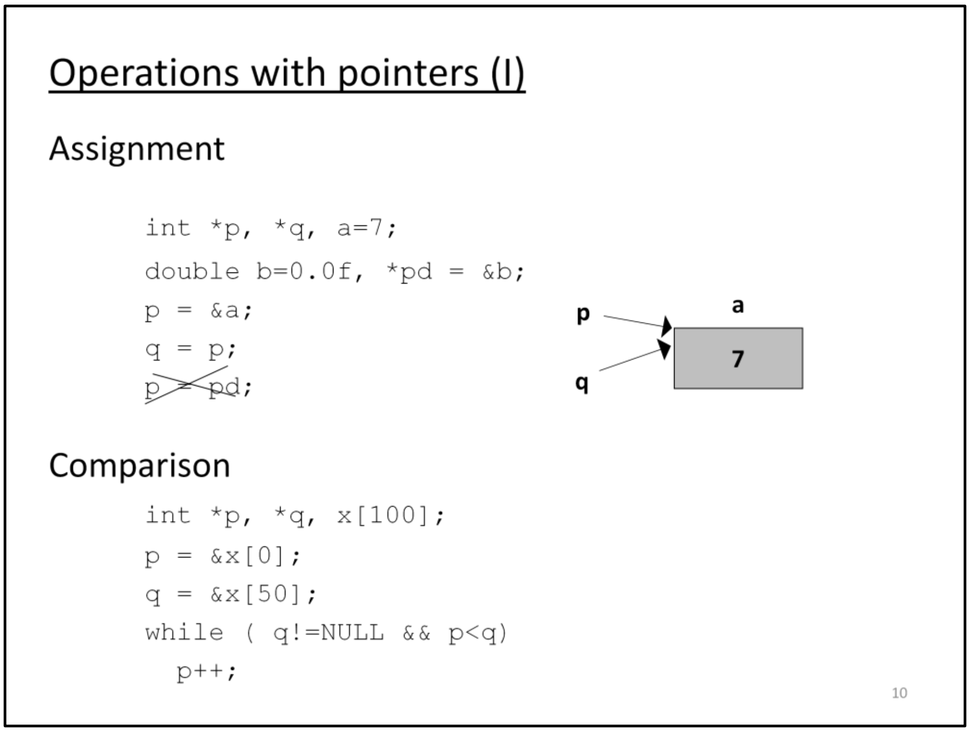
```
#include <stdio.h>

// definition of the type of structure

typedef struct date
{
  int day;
  int month;
  int year;
}tDate;

int main()
{
  tDate d1, *pdate;

  pdate = &d1;
  d1.day = 29; //access to his members  (1)
  pdate->day = 23;                      (2)
  scanf("%d", &pdate->month);
  scanf("%d", &(*pdate).year);          (3)

  //......

  return 0;
}
```

9

Next slide presents the use of pointers to structures. We have defined the new data type *tDate* as a type of structure. The fact that it is defined out of the main function allows the use of this type in all the functions of the program.

We use the operator point (**.**) to access the members of a structure variable (line named (1)).

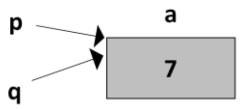
We use the operator arrow (**->**) to access the members of a structure which is pointed by a pointer (line named (2). Nevertheless the content of a pointer which points to a *struct* type is a structure. Therefore we can use the content to a pointer together with the operator point (**.**) to access the members of a structure (line named (3)).



Operations with pointers (I)

Assignment

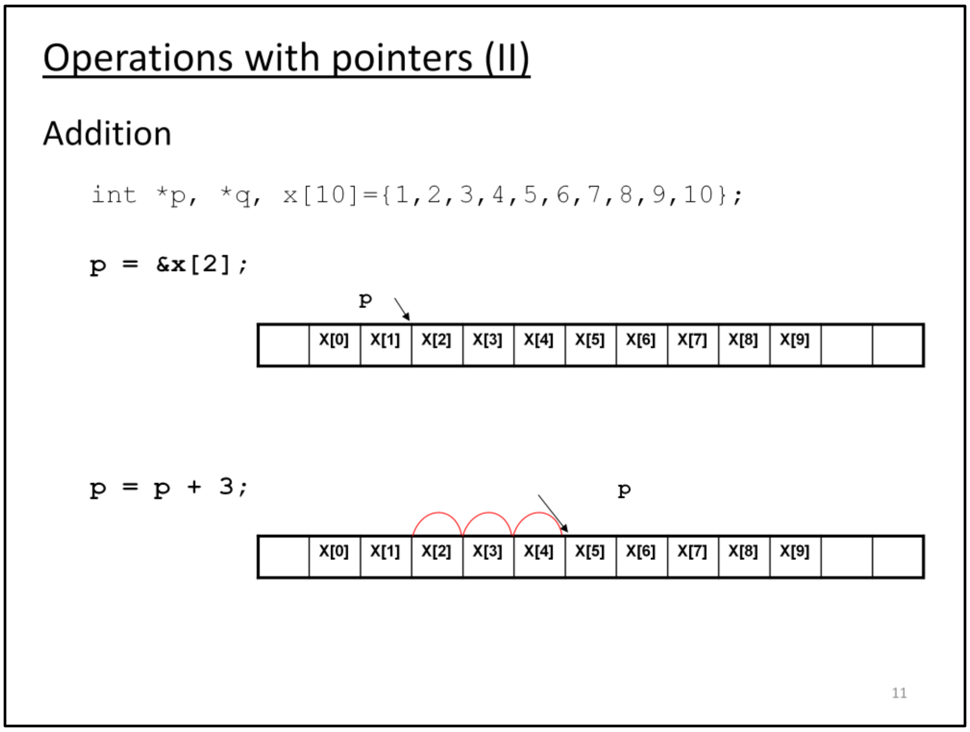
```
int *p, *q, a=7;
double b=0.0f, *pd = &b;
p = &a;
q = p;
~~p = pd~~;
```



Comparison

```
int *p, *q, x[100];
p = &x[0];
q = &x[50];
while ( q!=NULL && p<q)
  p++;
```

We can assign one pointer to other pointer if both pointers point to the same data type. The pointer *pd* cannot be assigned to the pointer *p* because they point to different data types. If we assign one pointer to another, both pointers will point to the same datum. Besides, two pointers can be compared with each other, and with an address.

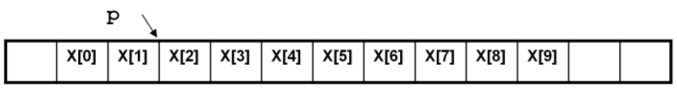


Operations with pointers (II)

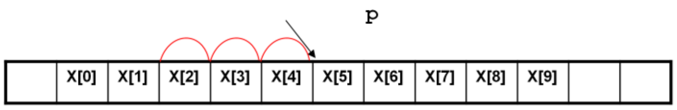
Addition

```
int *p, *q, x[10]={1,2,3,4,5,6,7,8,9,10};
```

```
p = &x[2];
```



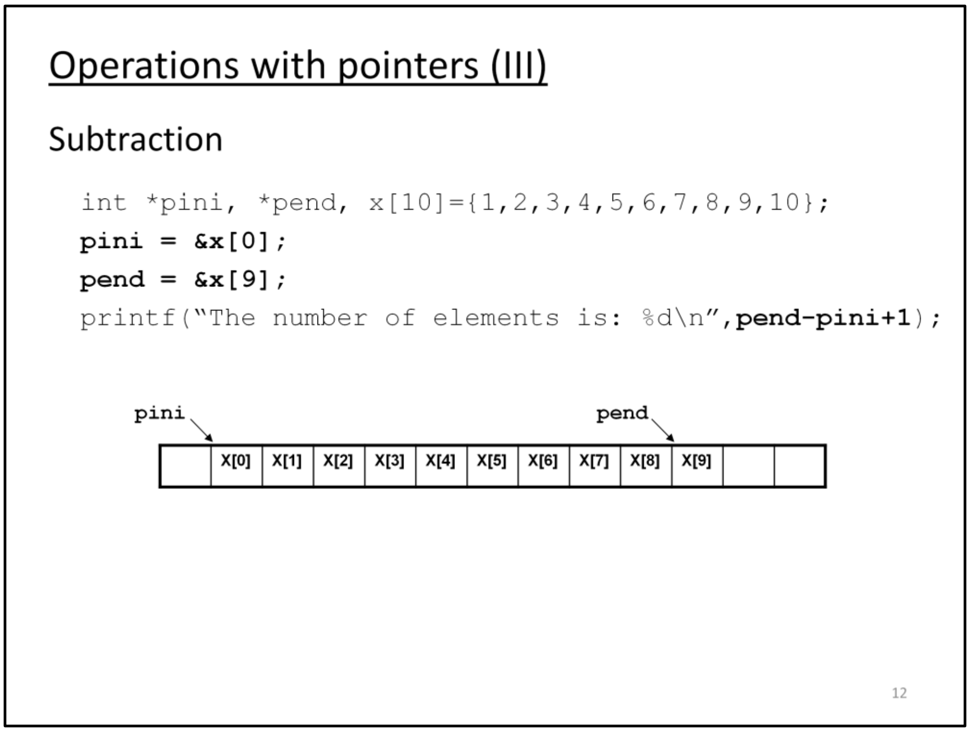
```
p = p + 3;
```



11

A number can be added to a pointer. The result of the operation is the pointer advances as many positions in the array, as the number added to it. For instance: we have a pointer which points to the third element of an array (we can see on the first operation). If we add *3* to the pointer *p*, the pointer advances *3* positions on the array and will point to the sixth element (*x[5]*).

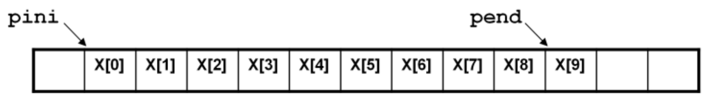
Therefore, adding a number to a pointer **DOESN'T MEAN** that you add the number to the address. In this case the pointer would point inside of the element *x[2]*, specifically on the three-quarters of the *x[2]*. **THIS CONCEPT IS WRONG**.



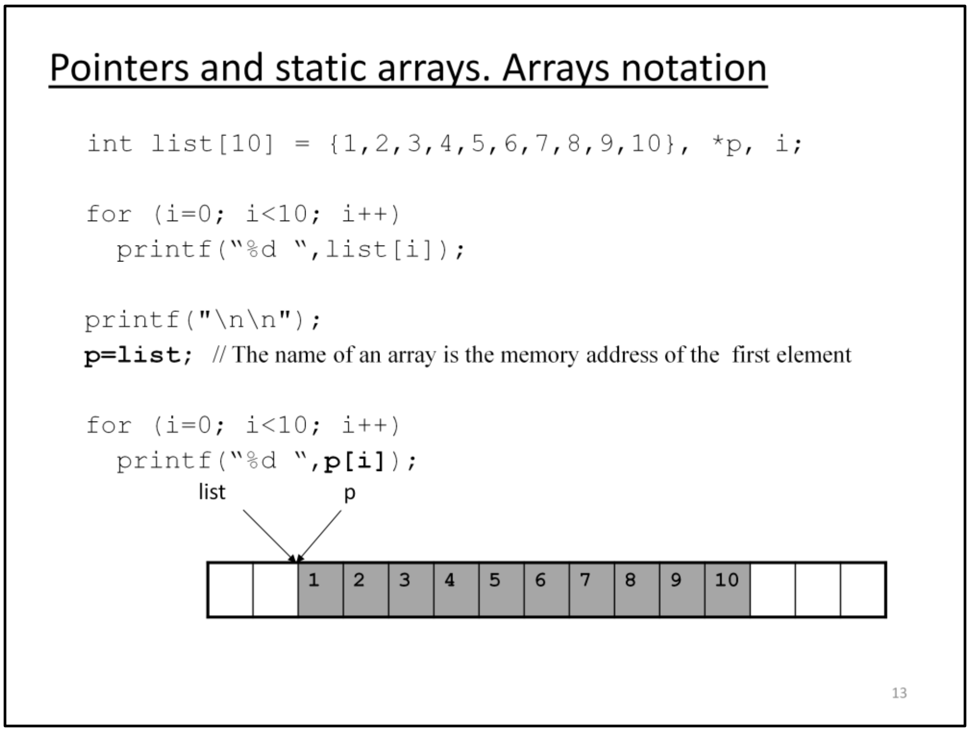
Operations with pointers (III)

Subtraction

```
int *pini, *pend, x[10]={1,2,3,4,5,6,7,8,9,10};
pini = &x[0];
pend = &x[9];
printf("The number of elements is: %d\n",pend-pini+1);
```

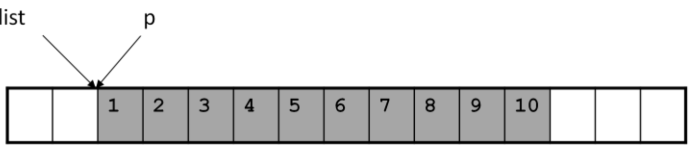


The same reasoning can be applied to the subtraction. Therefore, if we subtract two pointers we will obtain the number of elements that there are between them. The example prints the number of elements of the array.



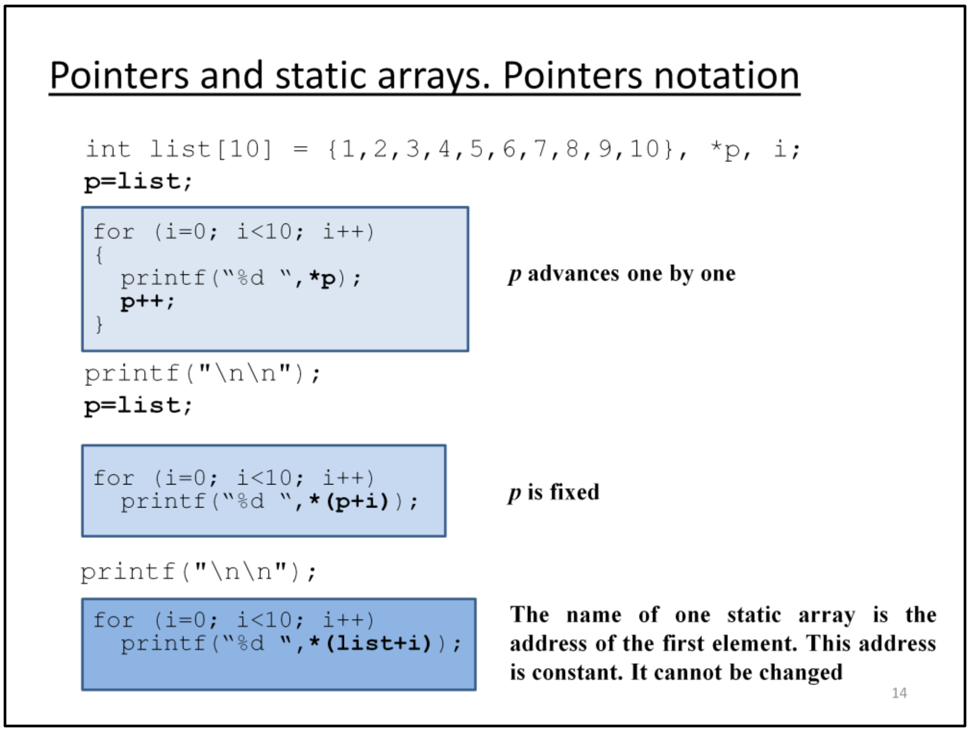
Pointers and static arrays. Arrays notation

```
int list[10] = {1,2,3,4,5,6,7,8,9,10}, *p, i;

for (i=0; i<10; i++)
  printf("%d ",list[i]);

printf("\n\n");
p=list;  // The name of an array is the memory address of the  first element

for (i=0; i<10; i++)
  printf("%d ",p[i]);
```



This slide shows the code for the display of a static array by following the notation of arrays. Firstly the array is printed. Then the pointer *p* is initialized to the beginning of the array and it is displayed again through the pointer. As we can see the way to access the content of the array is the same through the name of the array than through the pointer. This is because the name of the array is also a pointer which is pointing to the first element of the array. It has the characteristic that is a constant pointer. It cannot be modified.

By contrast, a pointer can change the value. That is, a pointer can point to a variable in a moment of the program and then it can be modified to point to other variable of the program.



Pointers and static arrays. Pointers notation

```
int list[10] = {1,2,3,4,5,6,7,8,9,10}, *p, i;
p=list;
```

```
for (i=0; i<10; i++)
{
  printf("%d ",*p);
  p++;
}
```

***p* advances one by one**

```
printf("\n\n");
p=list;
```

```
for (i=0; i<10; i++)
  printf("%d ",*(p+i));
```

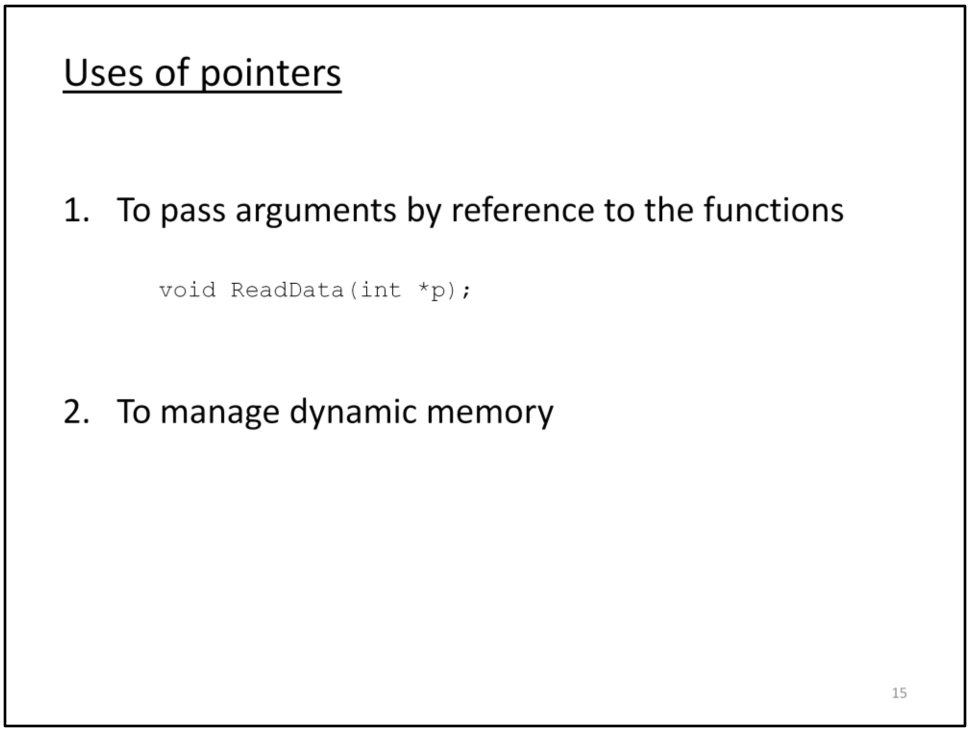
***p* is fixed**

```
printf("\n\n");
```

```
for (i=0; i<10; i++)
  printf("%d ",*(list+i));
```

The name of one static array is the address of the first element. This address is constant. It cannot be changed

In this case, instead of accessing the array with the square brackets, we use the content of the pointer and we advance it to the next position for the next iteration of the loop, (first box). The same can be made without moving *p* position, just by adding the index to *p* and extracting the content (second box). Finally, we use the name of the array for the last case. We cannot use it for the first case since the pointer is changing and, as we saw on the last slide, the name of an array cannot be modified (third box). This type of notation to access the information is called notation of pointers.



Uses of pointers

1. To pass arguments by reference to the functions

```
void ReadData(int *p);
```

2. To manage dynamic memory

15

The two uses of the pointers are:

1. To pass arguments by reference to the functions

If the *main()* function calls to another function and pass to it a variable by reference, the address of the variable must be passed. The function takes this address in a pointer and works with the content of the pointer. All of this means that the function can access to the variable that has been passed from *main()*. Therefore, the function can modify the variable received and the change affects to the variable of *main()*. For instance, if we have the next prototype and call for a function:

```
//...
void ReadData(int *p);


int main()
{
    int a;
    //...
    ReadData(&a);
    //...
}
```

The function *ReadData()* reads a value from keyboard and the variable *a* of *main()* takes this value. By contrast, if the variable is passed by value:

```
//...
void ReadData(int temp);
int main()
{
    int a;
    //...
    ReadData(a);
    //...
}
```

The value that the *ReadData()* function reads from keyboard is not assigned to the *a* variable of *main()* since *a* and *temp* are dos independent variables. Even, if both would have the same name.

```
//...
void ReadData(int a);
int main()
{
    int a;
    //...
    ReadData(a);
    //...
}
```

In this case the name of the variable in *main()* is *a* and the name of the variable in the function is also *a*. Nevertheless, the compiler recognizes the two variables as different and independent from each other since they belong to different functions.

Next slide shows a memory map that represents this concept

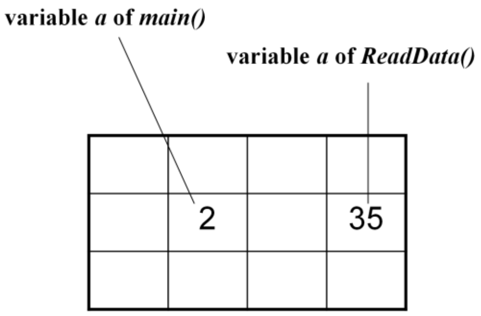
2. To manage dynamic memory

Slides 19 to 30 show theory and examples of dynamic memory.

Passing arguments by value. Memory map

```
//...
void ReadData(int a);
int main()
{
   int a=2;
   //...

   ReadData(a);
   printf("%d\n", a);
   //...
}

void ReadData (int a)
{
   a = 35
}
```

??

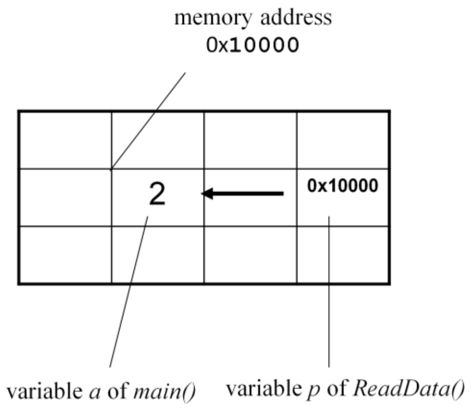


***printf()* result: 2**

Passing arguments by reference. Memory map(I)

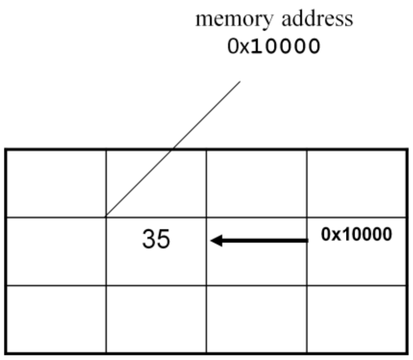
```
//...
void ReadData (int *p);

int main()
{
  int  a=2;
  ReadData (&a);
  printf ("%d\n",a); ??
  return 0;
}

void ReadData (int *p)
{
  scanf("%d", p);
}
```

user enters the value 35



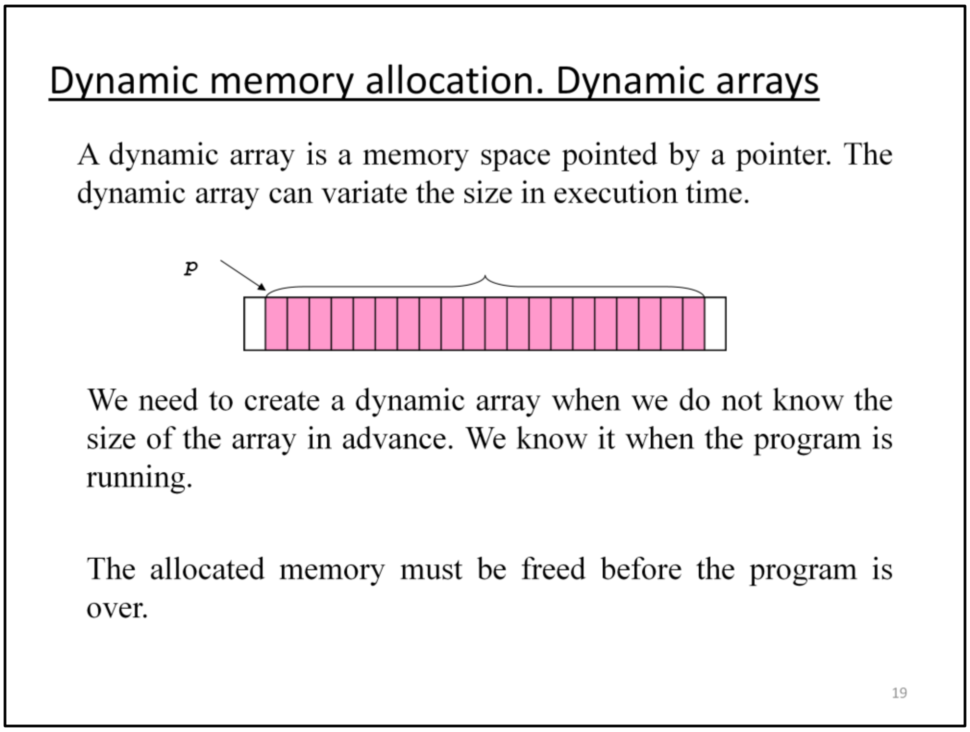
17

Passing arguments by reference. Memory map(II)



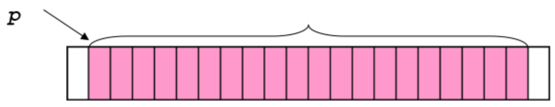
***printf()* result: 35**

Evolution of the previous graph



Dynamic memory allocation. Dynamic arrays

A dynamic array is a memory space pointed by a pointer. The dynamic array can variate the size in execution time.



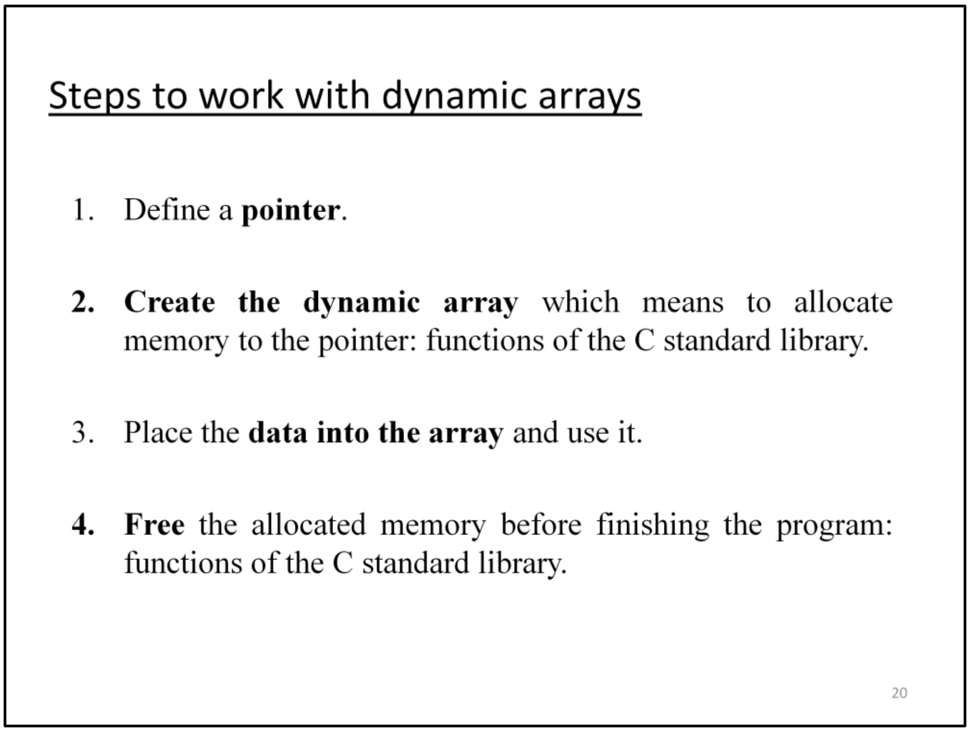
We need to create a dynamic array when we do not know the size of the array in advance. We know it when the program is running.

The allocated memory must be freed before the program is over.

A pointer can be initialized either by assigning the memory address of a variable of the program or by allocating dynamic memory.

We use a dynamic array to store data in memory. The difference with the static arrays is we are able to resize the array. There is not limit for number of elements, just the limit of the memory. Therefore, we can add as new elements as we need. We can also eliminate elements.

We have to free the memory we have allocated before finishing the execution of the program.



Steps to work with dynamic arrays

1. Define a **pointer**.

2. **Create the dynamic array** which means to allocate memory to the pointer: functions of the C standard library.

3. Place the **data into the array** and use it.

4. **Free** the allocated memory before finishing the program: functions of the C standard library.

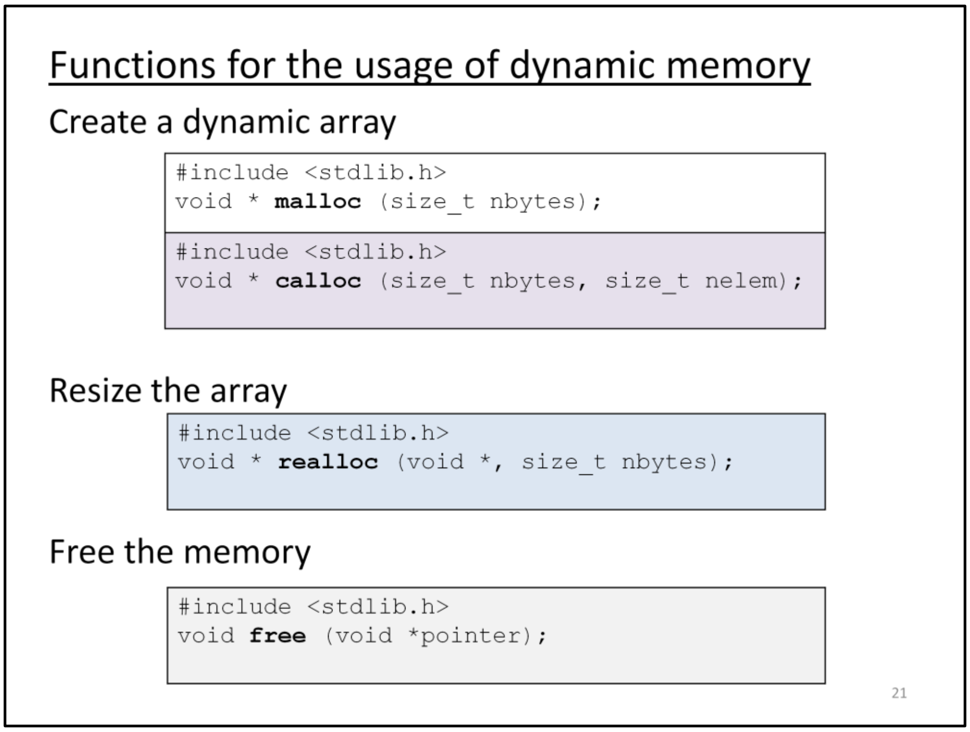
20

Firstly, we need a pointer pointing to the data type that we need store. It could be a pointer to *int* if we are going to store the number of students that are accepted in the University every year. It could be a pointer to *double* if we are going to store the average of the temperatures of each the year in Finland. It could be a pointer to *struct* if we wish to save the personal data of every student of Metropolia University.

Secondly, we allocate memory to this pointer for one or several elements through the specific functions of the C Language library that we are going to see next.

The third step is to read the data (usually from the keyboard) and to place them into the dynamic array, that is, in the memory we just to allocate. After that we can use the array as many times as we need.

Finally, we should free the memory before to finish the program.



Functions for the usage of dynamic memory

Create a dynamic array

```
#include <stdlib.h>
void * malloc (size_t nbytes);
```

```
#include <stdlib.h>
void * calloc (size_t nbytes, size_t nelem);
```

Resize the array

```
#include <stdlib.h>
void * realloc (void *, size_t nbytes);
```

Free the memory

```
#include <stdlib.h>
void free (void *pointer);
```

21

We can use *malloc* or *calloc* functions to create a dynamic array. Both functions carry out the same task. The differences between them are two:

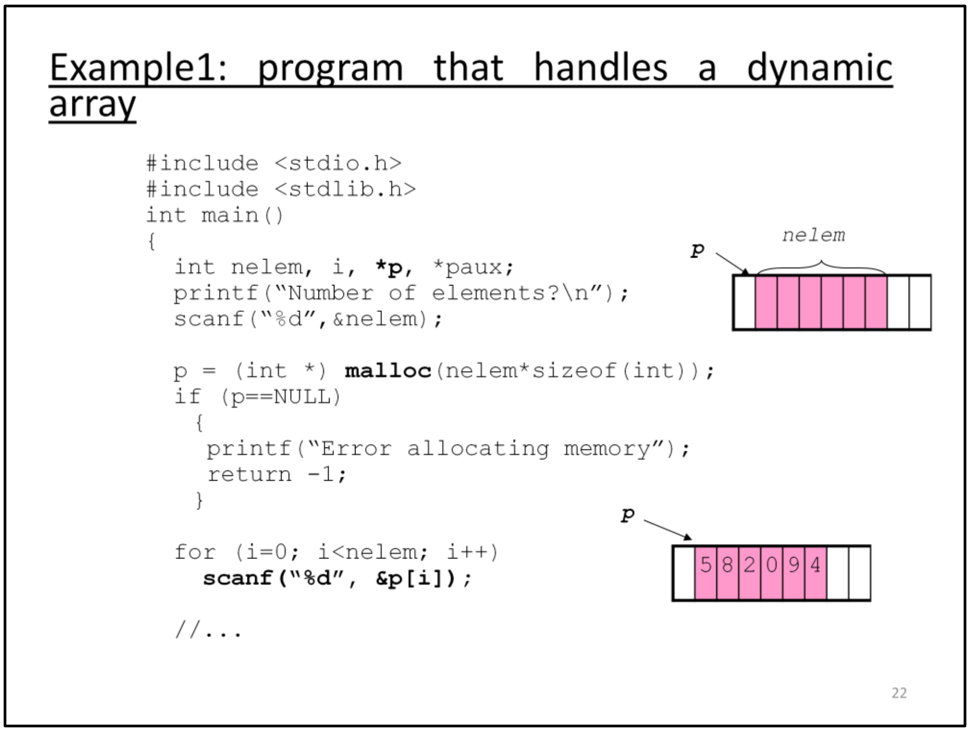
1. The number of parameters they receive. *Malloc* receives only one parameter which containing the number of bytes of the all elements to be allocated. We usually utilize *sizeof* (data type) multiplied by the number of elements of the array. *Calloc* receives the same information through two parameters: the number of elements of the dynamic array and the number of bytes that an element takes up.

2. *Calloc* initializes to zero the memory space allocated.

Both functions return a pointer pointing to the memory space allocated. The pointer is a pointer to a generic data (*void **). Therefore, we should convert this pointer to the specific type of pointer of the program, as done in the following example. If the memory cannot be allocated, both functions return *NULL*.

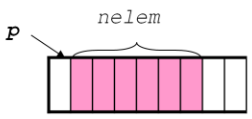
The *realloc* function allows us to add one or several elements at the end of the array. This function also allows us to eliminate elements from the end of the array. A pointer pointing to the original array and the total of bytes (after the extension) need to be passed to the function as parameters. This function returns a pointer pointing to the beginning of the extended array. If the extension was not possible to be made, the function will return *NULL*.

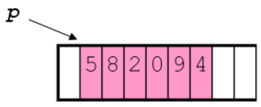
The *free* function releases the space of memory previously allocated with *malloc*, *calloc* or *realloc* which is pointed by the pointer that receives as parameter. A dynamic array only needs to be freed once, although the array had been expanded through the *realloc* function or the memory space is pointed by two pointers. A pointer which point to *NULL* mustn’t be freed.



Example1: program that handles a dynamic array

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int nelem, i, *p, *paux;
  printf("Number of elements?\n");
  scanf("%d",&nelem);

  p = (int *) malloc(nelem*sizeof(int));
  if (p==NULL)
   {
    printf("Error allocating memory");
    return -1;
   }

  for (i=0; i<nelem; i++)
    scanf("%d", &p[i]);

  //...
```

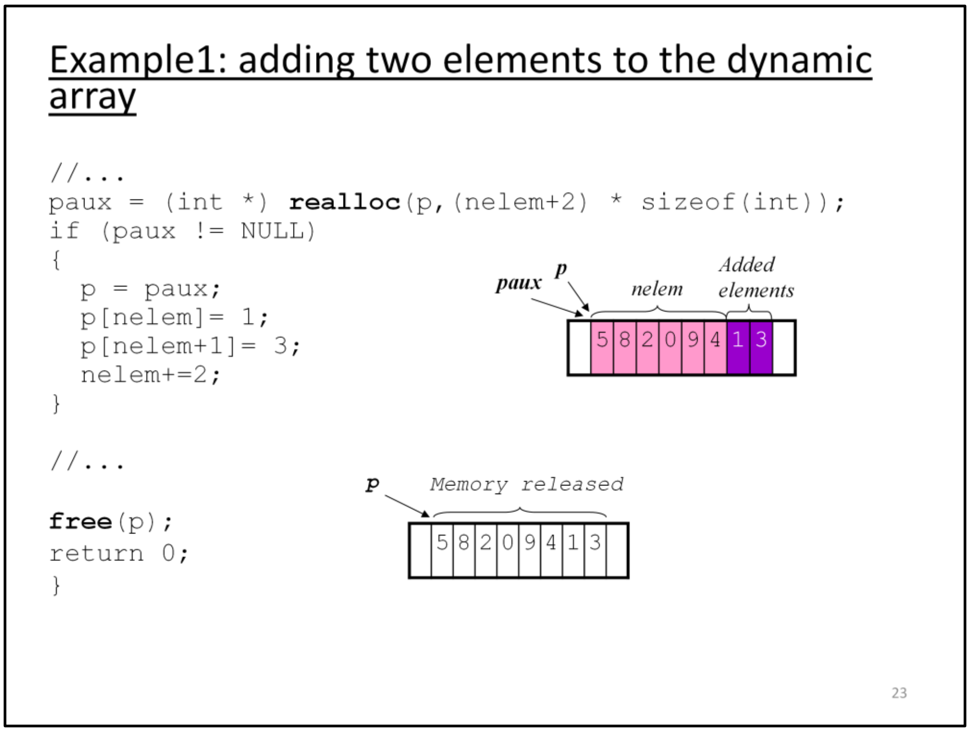




This example creates and initializes a dynamic array of integers. After that, it adds two elements to the array. Finally, it releases the memory before finishing the program.

The code follows the steps previously showed:

1. Defines the pointer (*p*) and reads from keyboard the number of elements that will have the array.
2. Allocates memory to the pointer through the *malloc* function. Passes to the function the number of bytes of the all elements. Everything is done as we have described before.
3. Reads the data from keyboard by using the *scanf* statement. Locates the data into the dynamic array.



Example1: adding two elements to the dynamic array

```
//...
paux = (int *) realloc(p,(nelem+2) * sizeof(int));
if (paux != NULL)
{
  p = paux;
  p[nelem]= 1;
  p[nelem+1]= 3;
  nelem+=2;
}

//...

free(p);
return 0;
}
```

The program continues and two elements are added at the end of the array through the *realloc* function.

As we described, *realloc* returns *NULL* if has problems allocating memory. So, we should use an auxiliary pointer (*paux*) in order not to lose our original data if an error happens. That is:

This statement is incorrect:

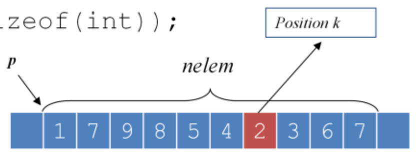
```
p = (int *) realloc(p,(nelem+2) * sizeof(int));
```

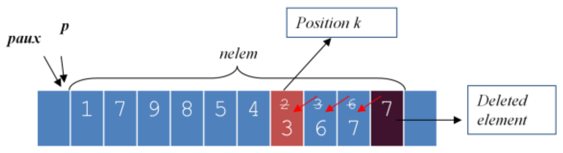
because if *realloc* return *NULL*, the pointer *p* which is pointing to the data, will point *NULL*, and then we cannot recover the original data. To solve the problem we use an auxiliary pointer (*paux*). In the case that *paux* is different of *NULL* we will assign *paux* to *p*. In this moment we will have two more spaces in our array that we can use.

Next step is to assign two data to the new elements. We write the number *1* and *3* on the last spaces of the array: *p[nelem]* and *p[nelemn+1]*. Finally we should update the total number of elements of the array. In the end we free the memory.

Example2: removing the element located in the position k

```
int nelem, *p,i,k;
printf("Number of elements?\n");
scanf("%d",&nelem);

p = (int *)malloc(nelem * sizeof(int));

//....

for (i=k; i<=nelem-2; i++)
  p[i]= p[i+1];
p = (int *) realloc(p,(nelem - 1) * sizeof(int));

nelem = nelem-1;

 // ...
```





24

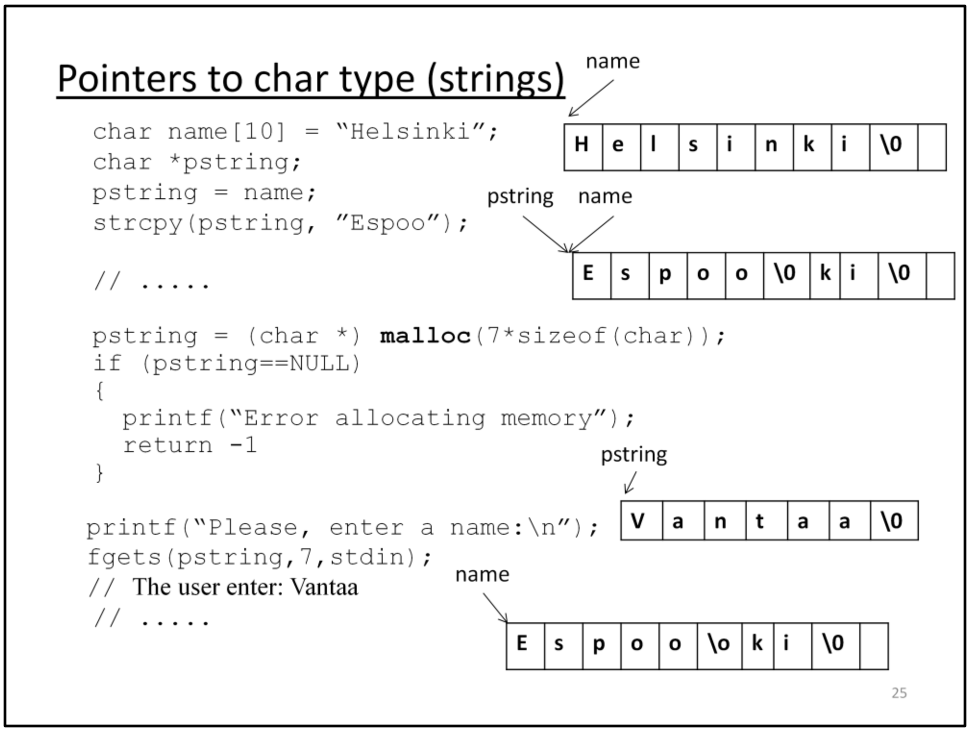
As you already know the first position of an array is 0 and the last one is n minus 1. In this example we remove one element of the array which is in one generic position, the position *k*. We are going to use the function *realloc* to this end. We should pass the *number of elements minus 1* as one of its arguments . Then *realloc* function will eliminate the last element that is *7*. We have a problem because we didn't want to eliminate the last one, but the element of the position *k* which value is *2*. We solve the problem by moving one position the data to the left side from the *k* position until the end of the array. We program a loop for this end and choose carefully the limits of the loop so that both don't pass the limits of the array and don't delete any useful datum.

The loop copies the number *3* on the position k in its first iteration. We have lost the number *2*. It doesn't matter. It is precisely that we wanted to do. The rest of the iterations of the loop moves the data to the left till the end of the array.

The last number is now duplicated. Therefore is the moment to call *realloc* and delete the last element. As a result we have the array without the element which value was *2*.

An auxiliary pointer is not needed in this example since we are not going to enlarge memory.

In case we would like to introduce a new element in the position *k*, firstly we should call to *realloc* to enlarge the array. Secondly we have to move the elements to the right, starting from the last position and finishing on the *k* position. Then we introduce the datum in the position *k*.



Pointers to char type (strings)

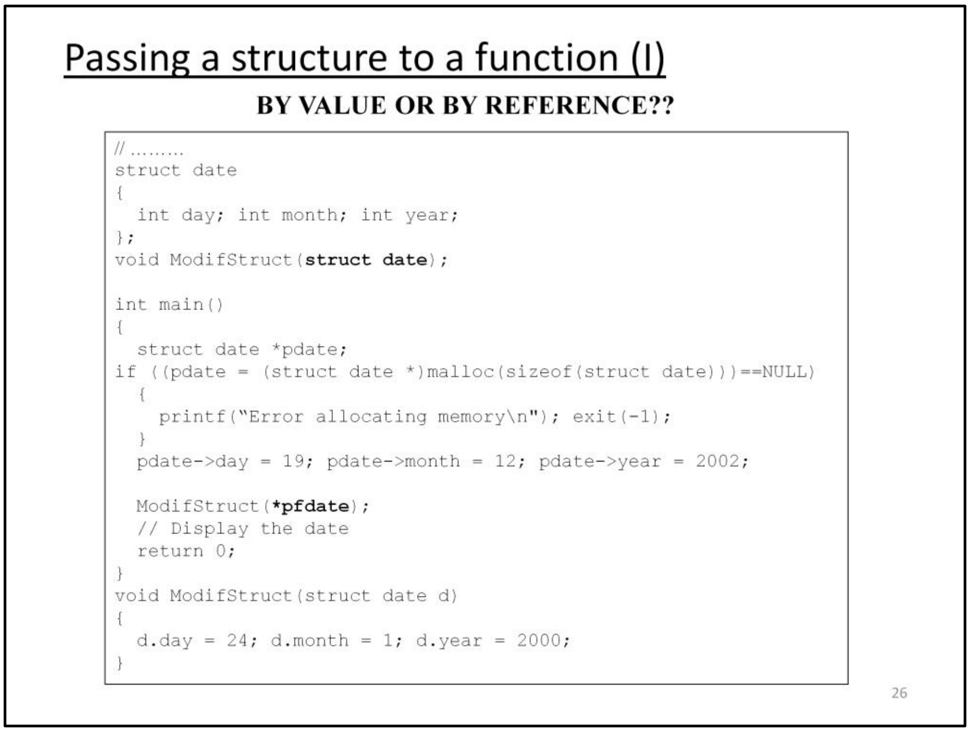
```
char name[10] = "Helsinki";
char *pstring;
pstring = name;
strcpy(pstring, "Espoo");

// .....

pstring = (char *) malloc(7*sizeof(char));
if (pstring==NULL)
{
  printf("Error allocating memory");
  return -1
}

printf("Please, enter a name:\n");
fgets(pstring,7,stdin);
// The user enter: Vantaa
// .....
```

name
H e l s i n k i \0
pstring name
E s p o o \0 k i \0
pstring
V a n t a a \0
name
E s p o o \o k i \0
25

We can also have pointers pointing to a *char* type. In this case we have *strings*. In the slide the pointer *pstring* firstly has been initialized to a *static string* and secondly dynamic memory has been allocated to it. In both cases we can use the functions of the *C* library for strings like *fgets()*, *strcpy()*, etc.

The slide presents three different stages:

1. The static array (*name*), initialized to the string *Helsinki*
2. The pointer *pstring* also points to the static array, and copy the string *Espoo* instead of *Helsinki*.
3. The pointer *pstring* points to the dynamic memory which has been allocated and it contains the string *Vantaa* that the user has introduced by keyboard.



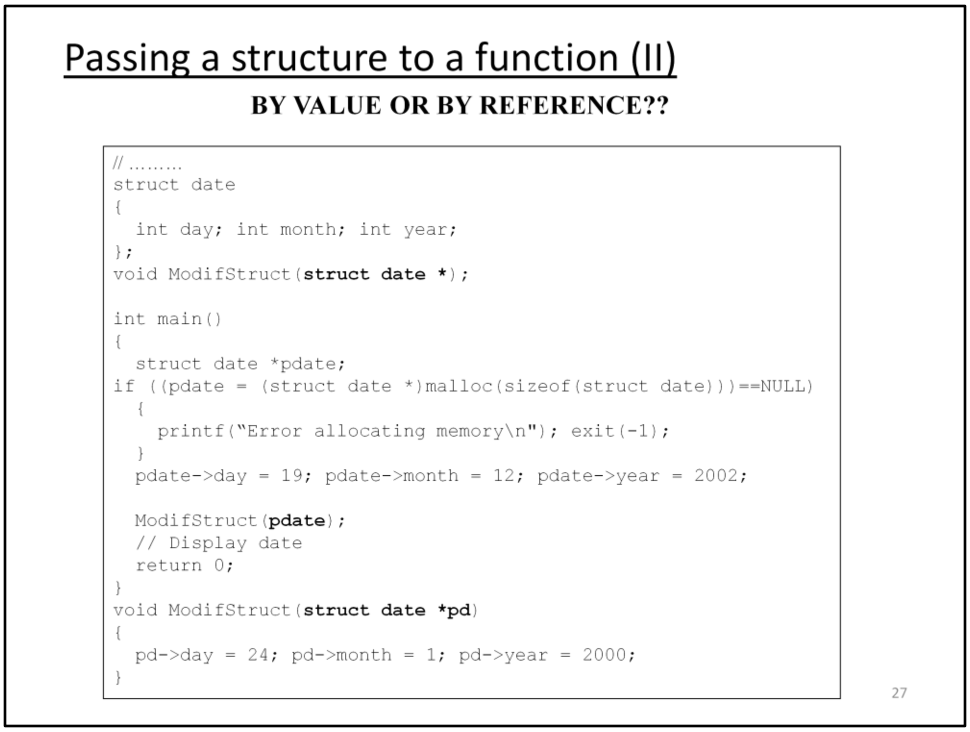
Passing a structure to a function (I)

BY VALUE OR BY REFERENCE??

```
// ………
struct date
{
  int day; int month; int year;
};
void ModifStruct(struct date);

int main()
{
  struct date *pdate;
if ((pdate = (struct date *)malloc(sizeof(struct date)))==NULL)
  {
    printf("Error allocating memory\n"); exit(-1);
  }
  pdate->day = 19; pdate->month = 12; pdate->year = 2002;

  ModifStruct(*pfdate);
  // Display the date
  return 0;
}
void ModifStruct(struct date d)
{
  d.day = 24; d.month = 1; d.year = 2000;
}
```

26

At this point we go back for a moment to the mode of passing arguments to the functions. Please could you **answer in the forum** how this function receives the parameter: **by value or by reference**?



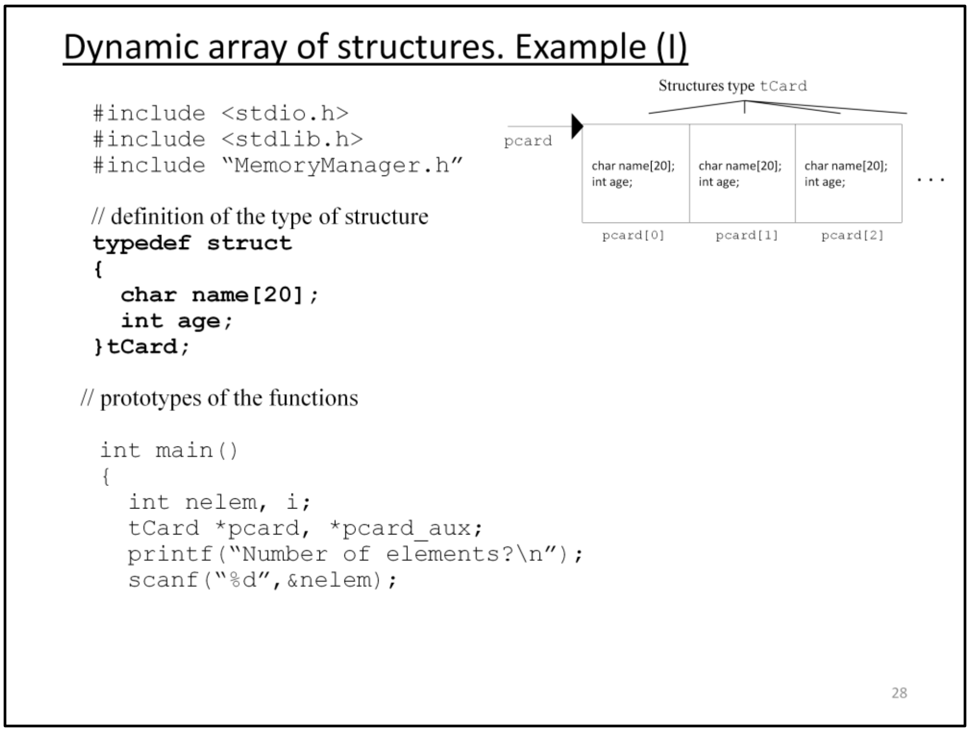
Passing a structure to a function (II)

BY VALUE OR BY REFERENCE??

```
// ………
struct date
{
  int day; int month; int year;
};
void ModifStruct(struct date *);

int main()
{
  struct date *pdate;
if ((pdate = (struct date *)malloc(sizeof(struct date)))==NULL)
  {
    printf("Error allocating memory\n"); exit(-1);
  }
  pdate->day = 19; pdate->month = 12; pdate->year = 2002;

  ModifStruct(pdate);
  // Display date
  return 0;
}
void ModifStruct(struct date *pd)
{
  pd->day = 24; pd->month = 1; pd->year = 2000;
}
```

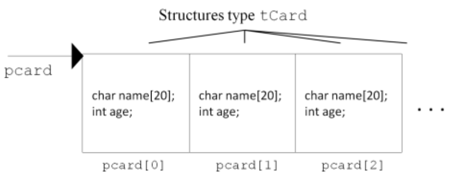
27

Please could you also **answer in the forum** how this function receives the parameter: **by value or by reference**?



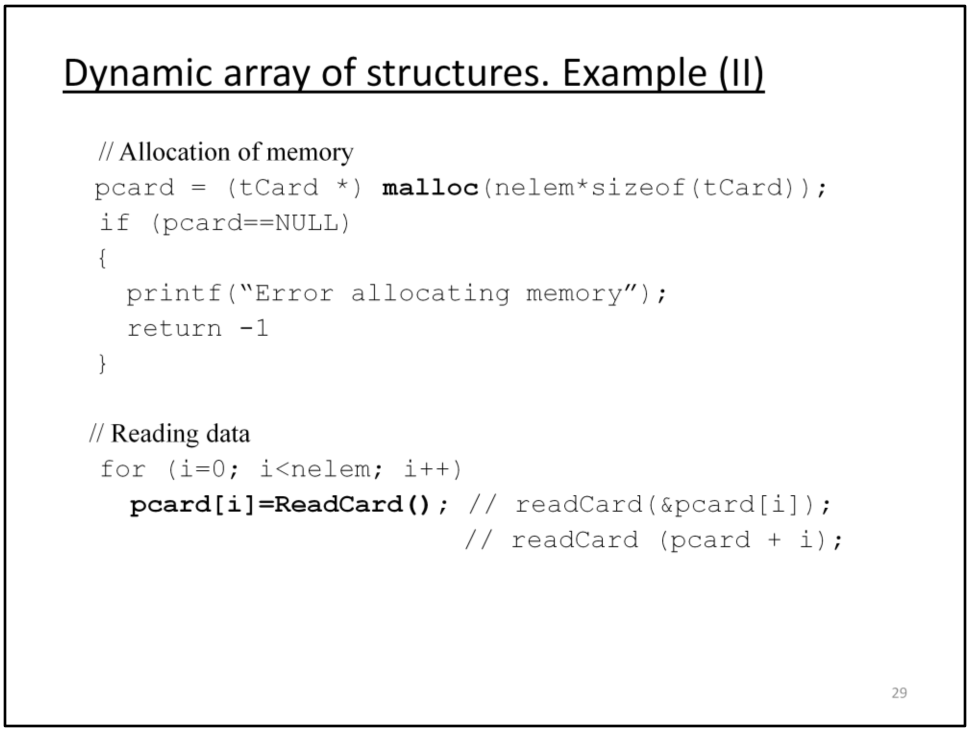
Dynamic array of structures. Example (I)

```
#include <stdio.h>
#include <stdlib.h>
#include "MemoryManager.h"

// definition of the type of structure
typedef struct
{
  char name[20];
  int age;
}tCard;

// prototypes of the functions

int main()
{
  int nelem, i;
  tCard *pcard, *pcard_aux;
  printf("Number of elements?\n");
  scanf("%d",&nelem);
```



28

Next example shows the use of a dynamic array of structures. We have a pointer to the structure *tCard* and we allocate dynamic memory to that pointer through the *malloc* function. We read the data elements from keyboard through the function *ReadCard()*. Then we add a new element and finally we free memory.



Dynamic array of structures. Example (II)

```
// Allocation of memory
pcard = (tCard *) malloc(nelem*sizeof(tCard));
if (pcard==NULL)
{
  printf("Error allocating memory");
  return -1
}

// Reading data
for (i=0; i<nelem; i++)
  pcard[i]=ReadCard(); // readCard(&pcard[i]);
                       // readCard (pcard + i);
```

29

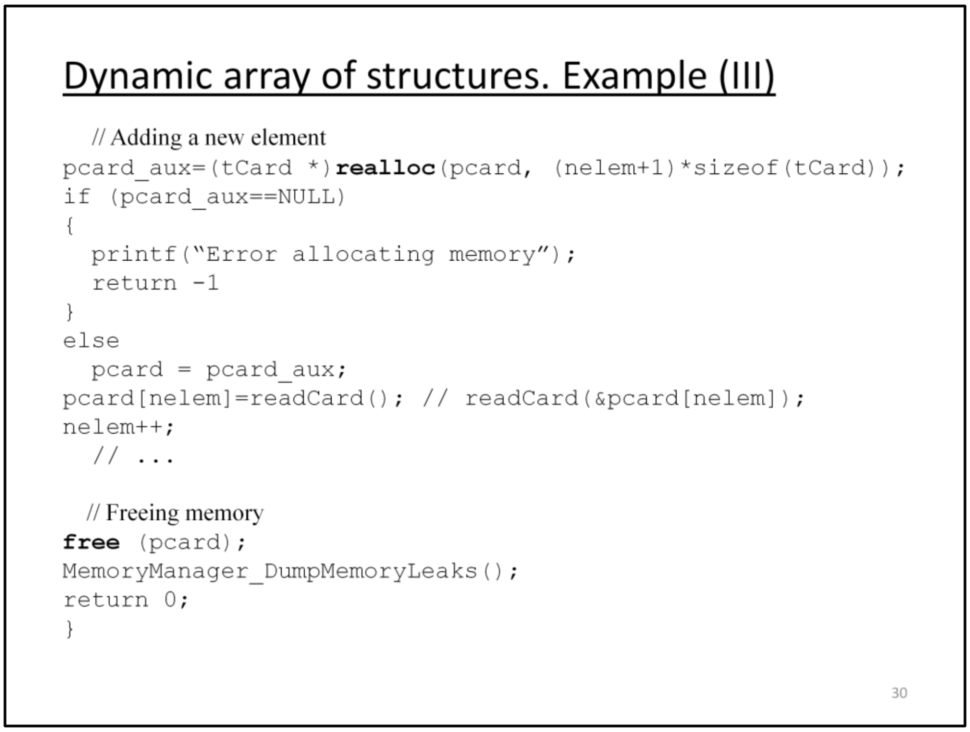
Practice by programing the function *ReadCard()* two times:

- First time using the prototype: *tCard ReadCar();*

In this case the function reads the data from keyboard and returns the structure.

- Second time using this other prototype: *void ReadCard(tCard *);*

where the structure is passed by reference in order the changes made to the structure inside the function (that is the reading from keyboard of its members) apply directly in the structure that it was passed as argument.



Dynamic array of structures. Example (III)

```
   // Adding a new element
pcard_aux=(tCard *)realloc(pcard, (nelem+1)*sizeof(tCard));
if (pcard_aux==NULL)
{
  printf("Error allocating memory");
  return -1
}
else
  pcard = pcard_aux;
pcard[nelem]=readCard(); // readCard(&pcard[nelem]);
nelem++;
  // ...

   // Freeing memory
free (pcard);
MemoryManager_DumpMemoryLeaks();
return 0;
}
```

30

Notes that this exercise incorporates the library *MemoryManager* that we should include it with all the programs we develop using dynamic memory (including the exam). This library checks the correct use we did of the dynamic memory since the compiler doesn't show some problems that can arise. The library shows it.

You can see how to use this library in the document *How to use MemoryManager library*.