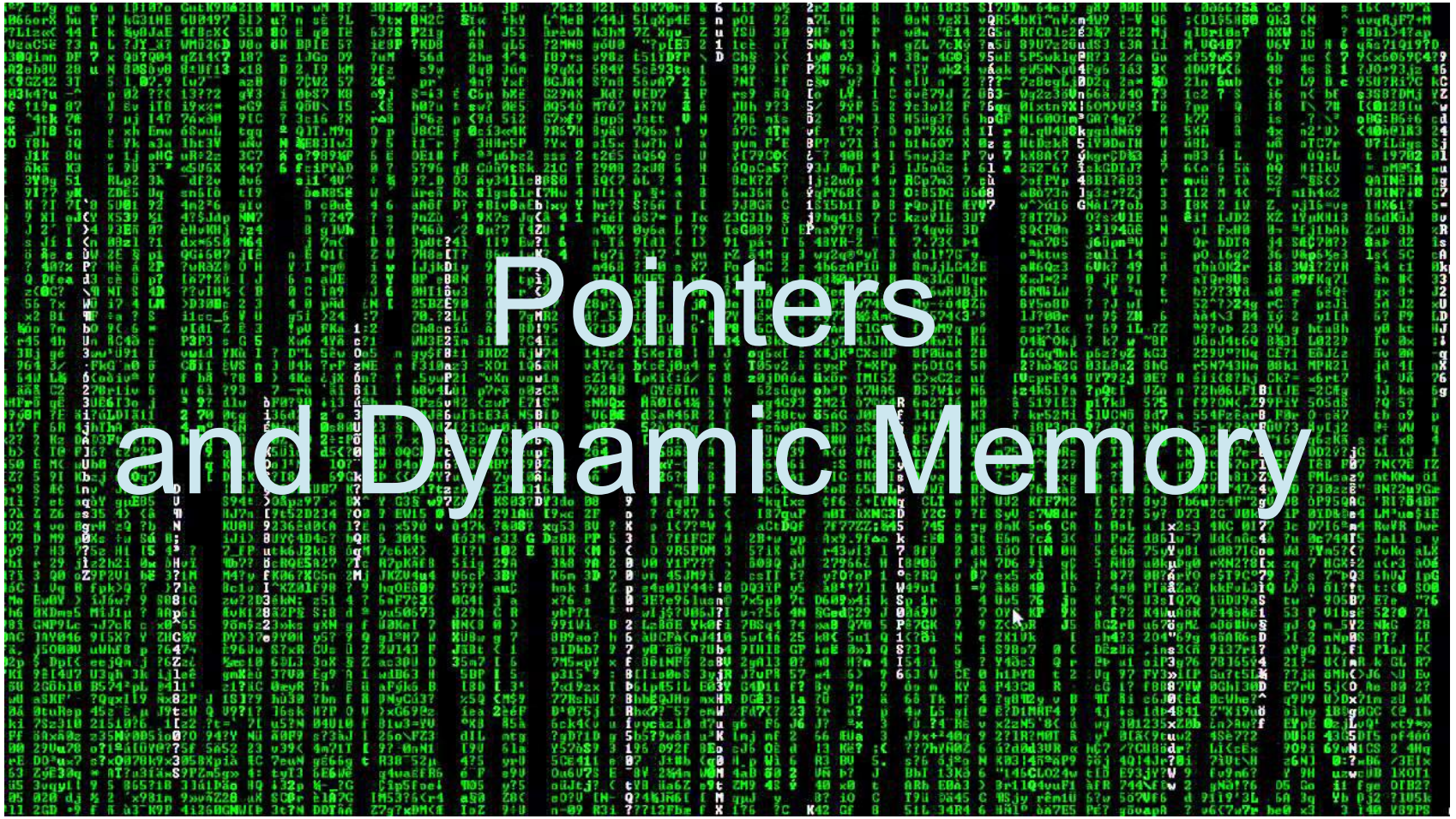
ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

C Programming



Pointers and Dynamic Memory

Ángel Perles
angel.perles@upv.es
R02

Contents

Objectives
Introduction
What's a pointer
Arithmetic operations
Arrays and pointers
Pointers in function parameters
Dynamic memory
Structures and pointers
Pointers: C vs Java

Objectives

- Declare, initialise and dereference pointers using * and &, and draw the memory diagram of a short code fragment
- Use pointer arithmetic to traverse arrays, knowing that the step size depends on the pointed type
- Write functions that modify the caller's data or receive arrays through pointers, using const when the data must not change.
- Allocate, use and release dynamic memory with *malloc()* and *free()*, checking for *NULL*
- Access structure members through a pointer with ->
- Recognise typical pointer bugs (uninitialised pointers, out-of-bounds access, memory leaks, use after free) and explain why C does not catch them as Java does

Introduction

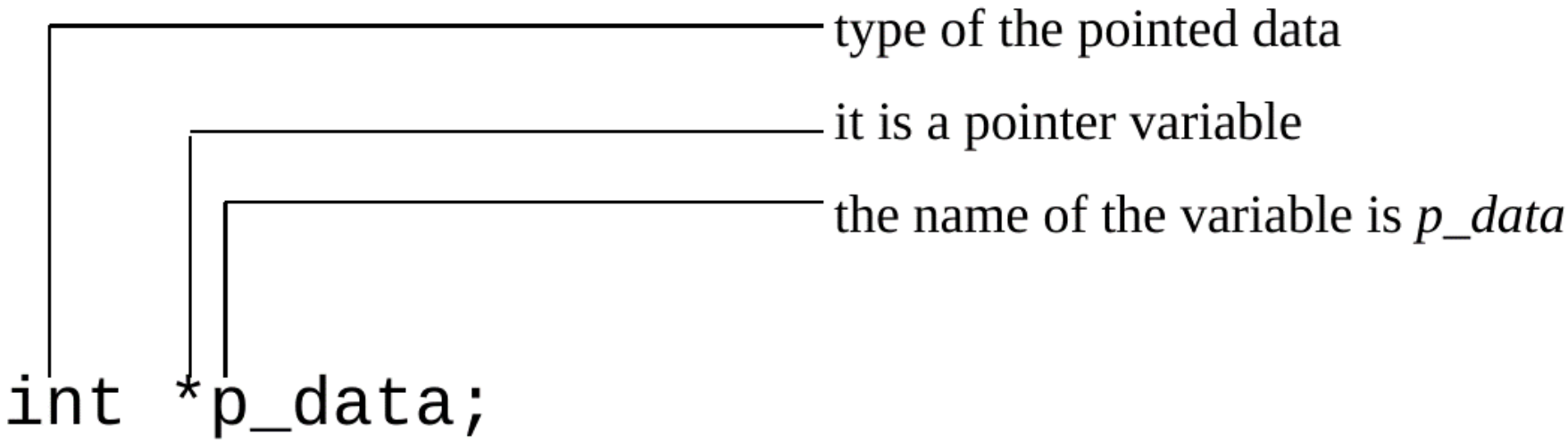
- Developing basic applications without **pointers** is feasible (lying, basically)
- In any case, C/C++ and **pointers** are intimately connected, so it is necessary to know them if we want to go further
- Using pointers is very error prone, and a source of security issues
- So, other programming languages such as Java hide them behind references
- Open your mind, and this will be easy



WHAT'S A POINTER

What's a pointer

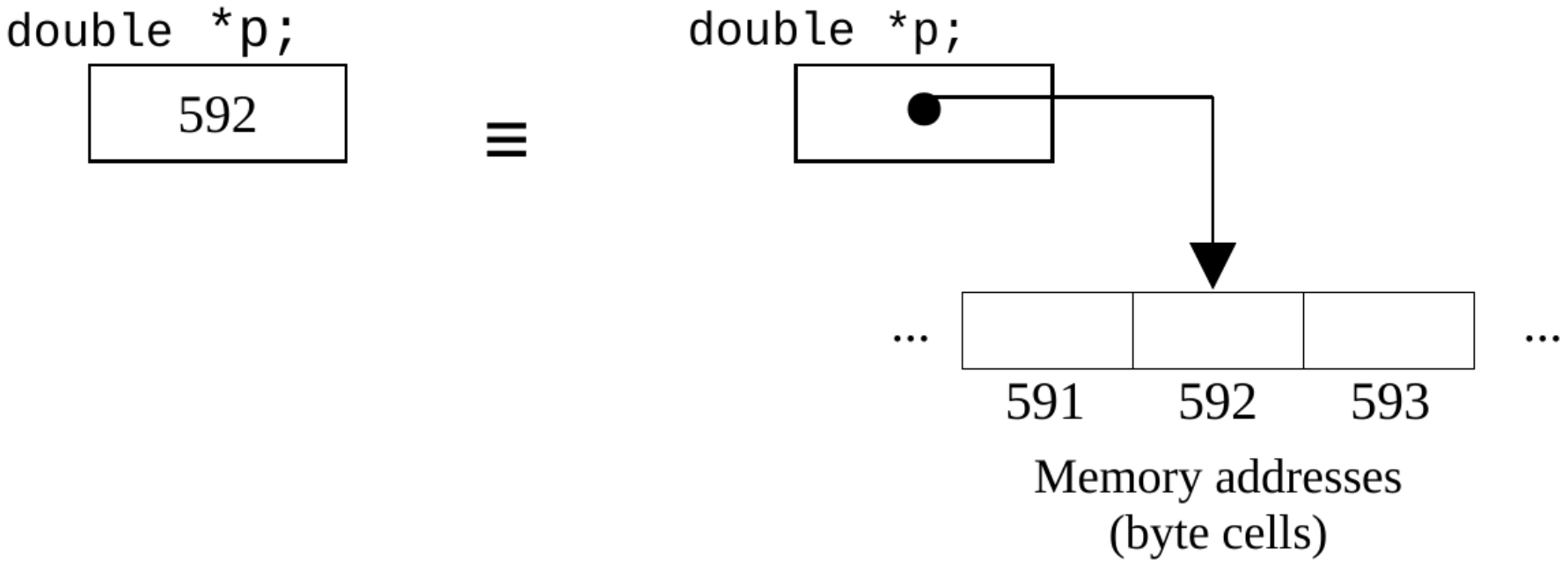
- A pointer is a **reference to a memory address**
- We can use variables to contain these references to memory, for example



- Here, the data type means how to handle the pointed data (an *int*, *double*, …)
- The *p_* prefix is not coincidental (Barr-C)

What's a pointer

- In jargon, the memory address contained in a pointer variable and the pointed address are synonyms



- The special value **NULL** means "no valid address"

What's a pointer

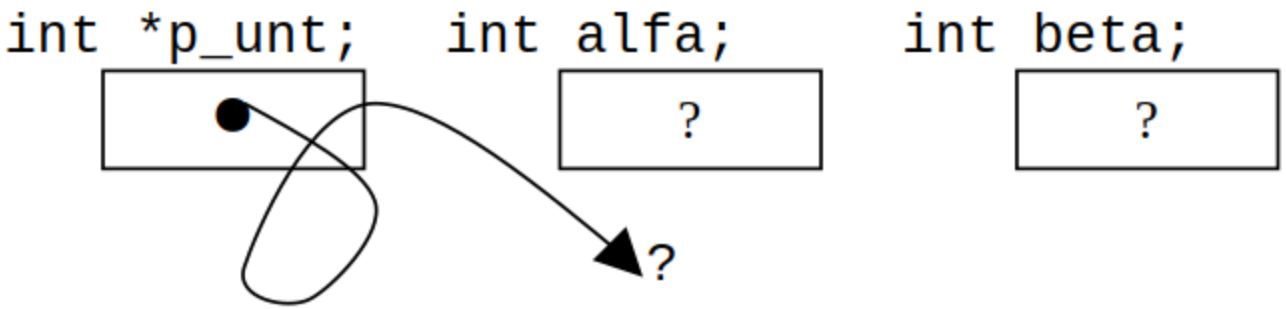
- Two key operators to use pointers are ***** and **&**
  - ****p_data*** means: "the content pointed to by the pointer *p_data*"
  - ***&name*** means: "the memory address of the name object (for example, a variable)
- Let's use an example to illustrate them (note * means two things here)

```
int main (void)
{
    int *p_unt;
    int alfa, beta;

    alfa   = 5;          // 1
    p_unt  = &alfa;      // 2
    beta   = *p_unt;     // 3
    *p_unt = 7;          // 4
}
```

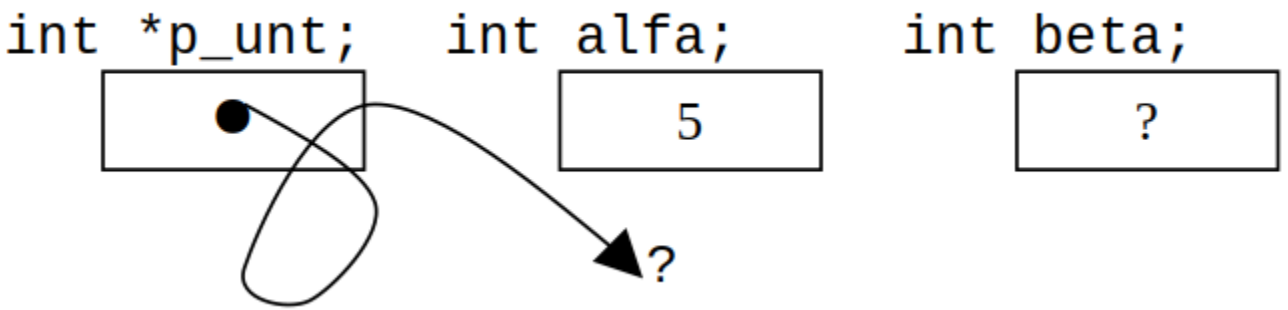
What's a pointer

- Let's use an example to illustrate them (cont'd…)
  - When entering the main() function, 3 automatic local variables are created
  - The content of them will be rubbish



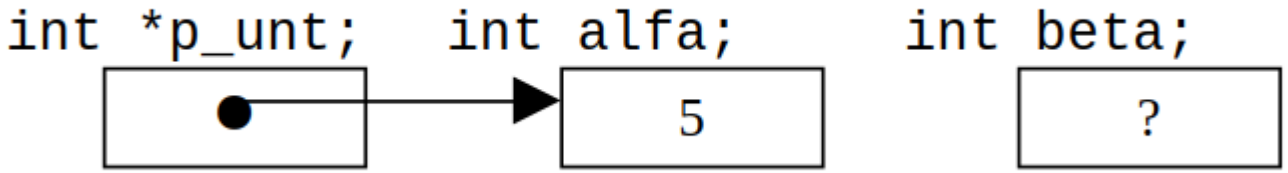
What's a pointer

- Let's use an example to illustrate them (cont'd…)
  - After running //1 we have



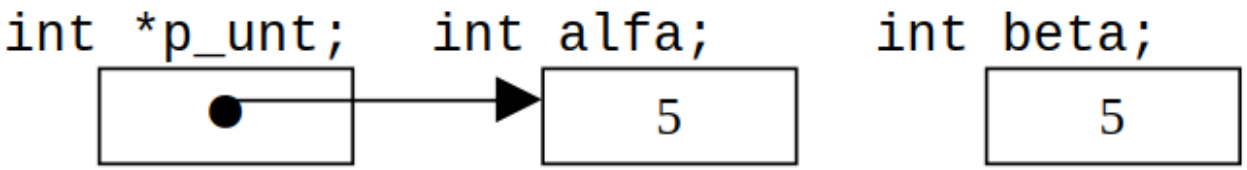
What's a pointer

- Let's use an example to illustrate them (cont'd…)
  - After running //2, *p_unt* points to the memory address containing *alfa*



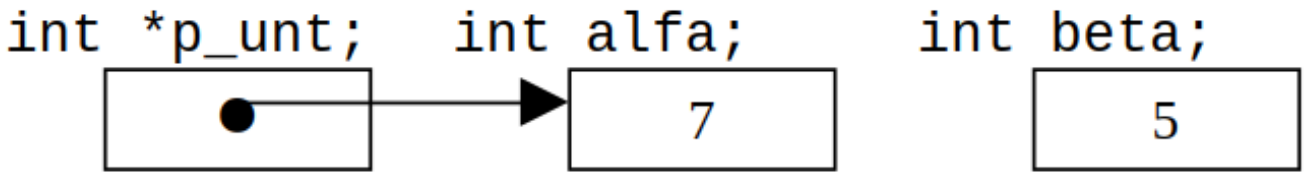
What's a pointer

- Let's use an example to illustrate them (cont'd…)
  - After running //3, *beta* is asigned the content pointed by *p_unt*



What's a pointer

- Let's use an example to illustrate them (cont'd…)
  - Finally, after running //4, the memory space pointed by *p_unt* is assigned the integer value 7



What's a pointer

- Activity: Write the C code in which you define 4 variables of type *double*. Define 4 pointers and set them to point to each of the 4 variables. Using these pointers, calculate the sum *a1 + a2 + a3* and store it in *result*.

```
double a1;
double a2;
double a3;
double result;
```



What's a pointer

- Activity: Do the same, but this time you can only declare two pointers, and you must carry out the entire operation using only pointers, without explicitly referring to a1, a2, a3 or the result; you may only use their memory addresses.
- The entire operation cannot be carried out in one go; you will need to carry out several steps and accumulate intermediate values as you go.

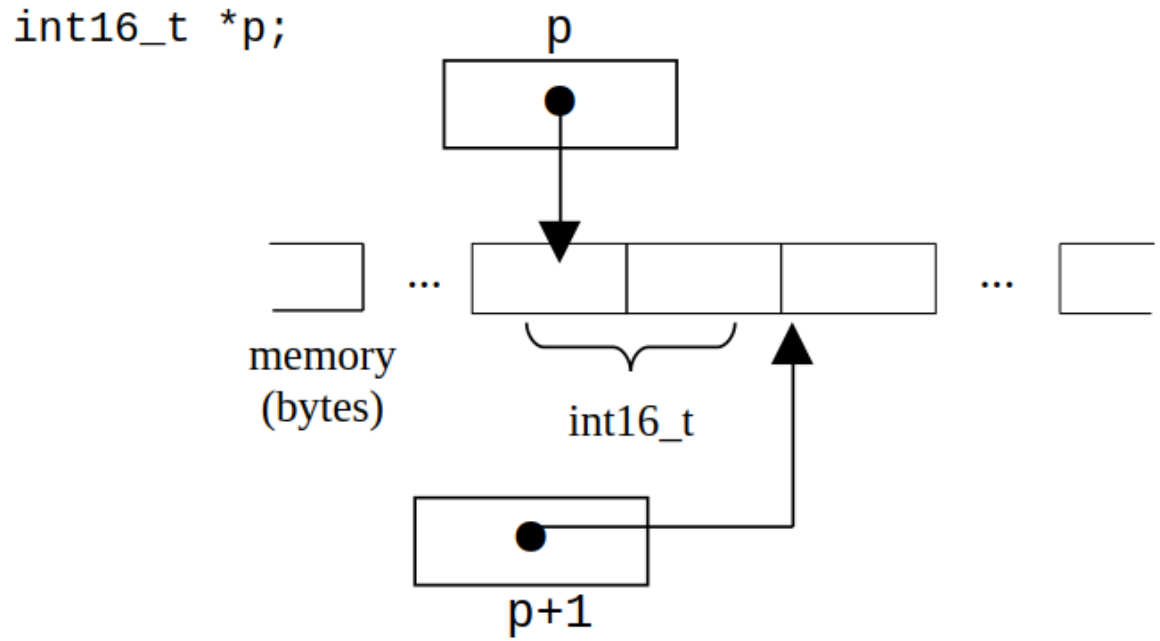
```
double a1;
double a2;
double a3;
double result;
```



ARITHMETIC OPERATIONS

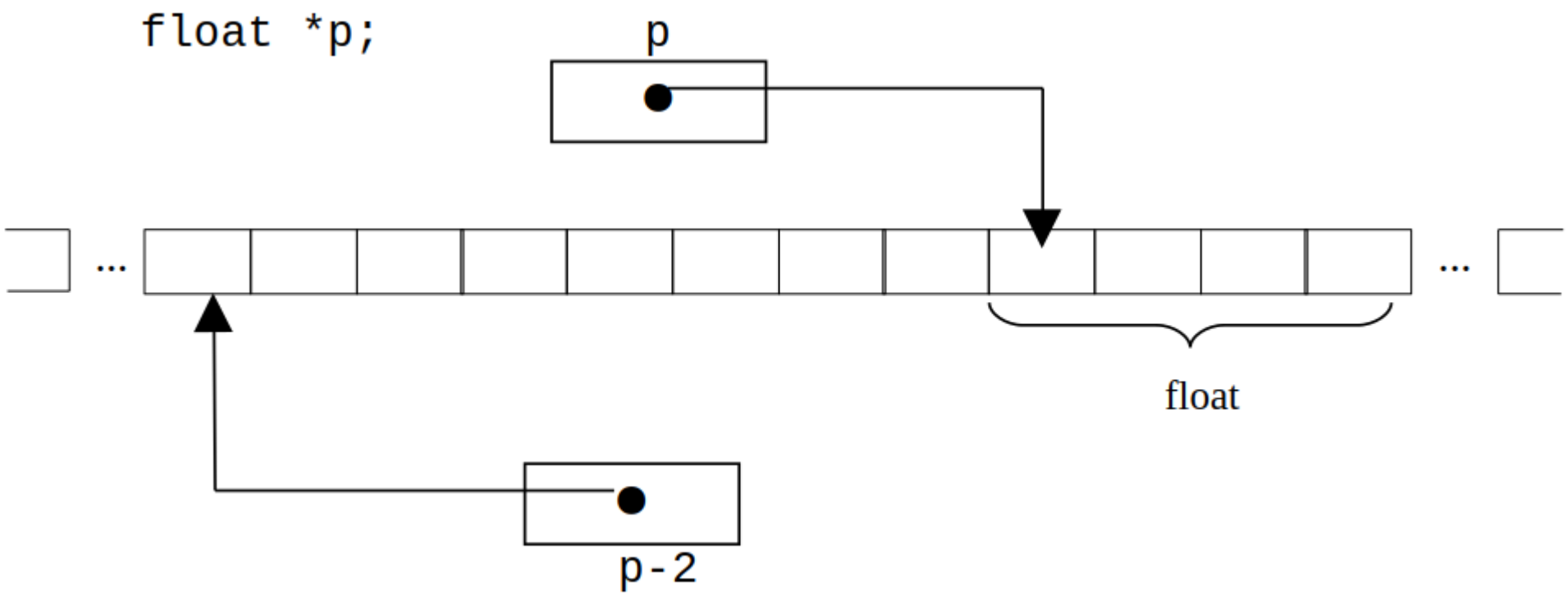
Arithmetic operations

- You can use arithmetic operators on pointers (+, -, ++, --, ...)
- This is equivalent to dancing with the pointer (moving forward or backwards)
- The size of the dance step is the number of bytes of the type of the pointer
- E.g., let's illustrate the idea assuming the size of an *int16_t* is 2 bytes



Arithmetic operations

- Now assume a float of size 4 bytes, subtracting 2 is equivalent to



Arithmetic operations

- Two examples. The right one scans memory till it finds an *int* with value -3525 (and, probably, causes a segmentation fault trap in an OS, but is silent in a microcontroller)

```
int *p;
int alfa;

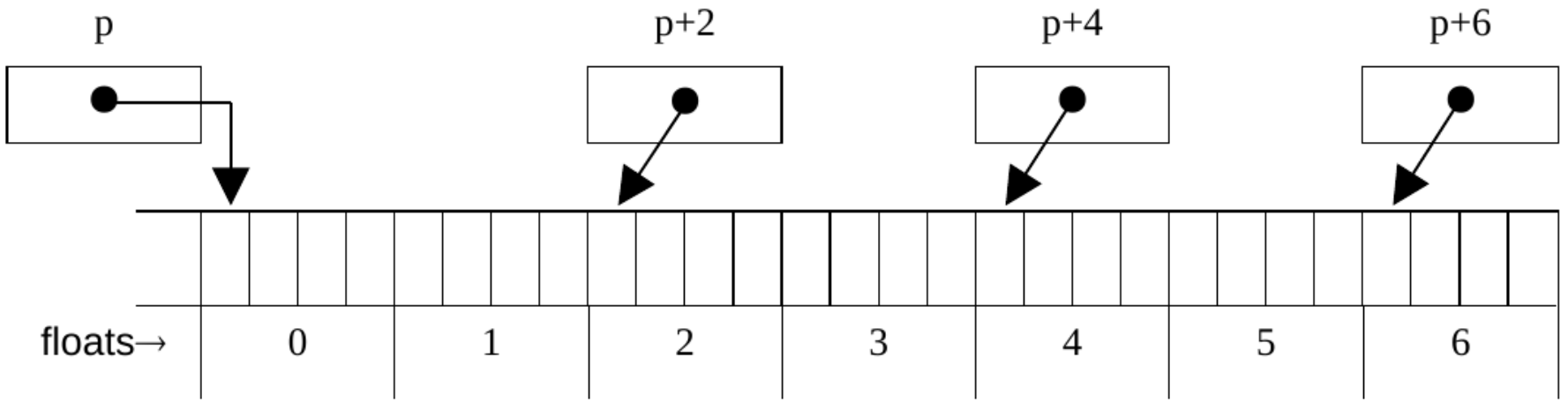
p    = ??;        //to some place
alfa = *(p+5);
```

```
int *p;

p = ??; //point to some place
while (*p != -3525)
{
    p++;
}
```

Arithmetic operations

- Activity: Store in a variable of type *double* the sum of 100 alternate *float* values, starting at *p*., given *float data[200]; float *p = data;*
- For example, assuming that a *float* occupies 4 bytes, the aim is to access the elements shown in the figure below.





Arithmetic operations

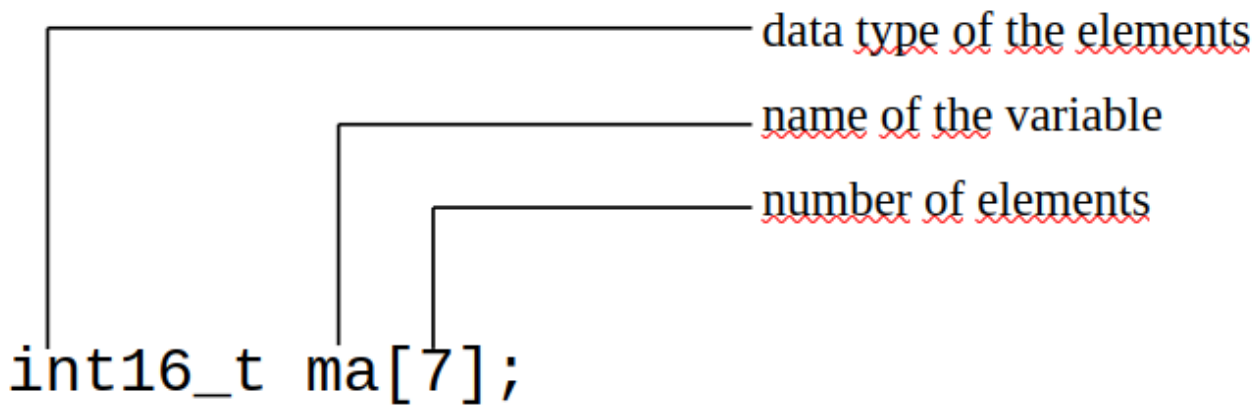
- IMPORTANT: Remember that the size of a data type could depend on the type of processor, the operating system, the compiler, etc. To avoid issues, always use the *sizeof(data type)* operator to find out how many bytes the data type occupies.
- For example, the following line in the Keil compiler for an 8051 microcontroller will return 2, and 4 for the Arm Cortex-M4 microcontroller. On an Android smartphone (Linux), GCC compiling this line will return 4 for both 32-bit and 64-bit architectures… maybe.

```
printf("Here, the size of an int is %zu\n", sizeof(int));
```

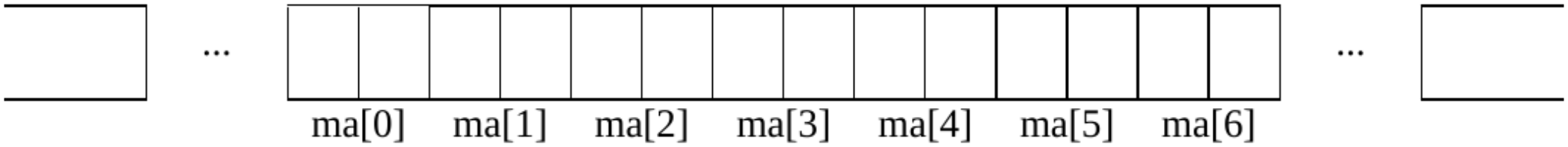
ARRAYS AND POINTERS

Arrays and pointers

- An array in C is a collection of elements of the same type arranged sequentially in memory
- For example, given the following declaration of a C array



- Assuming 2 bytes per element, we have the following memory allocation

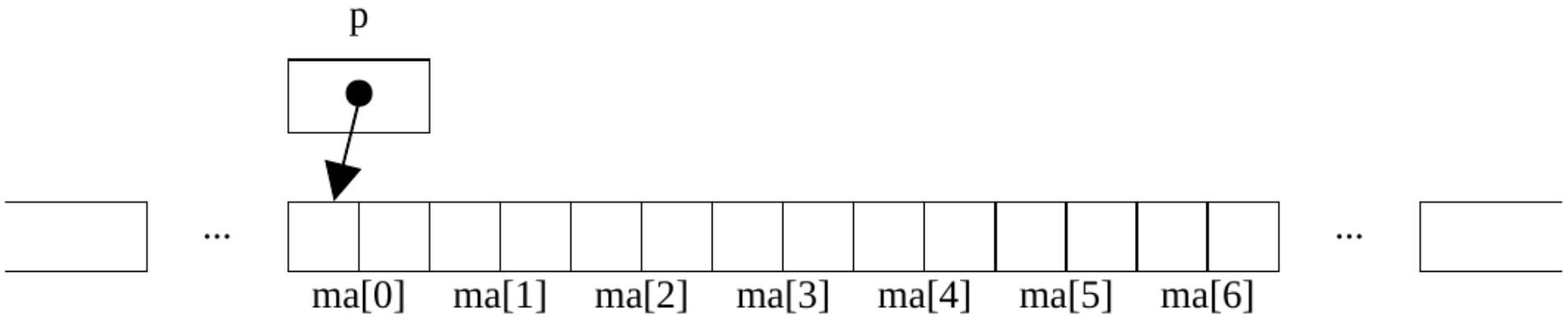


Arrays and pointers

- We can use a pointer to access the elements. In fact, it's quite common to do it this way. E.g.

```
int16_t *p;
int16_t ma[7];

p = &ma[0];
```



Arrays and pointers

- For the previous example, we could access element *ma[2]* in different ways

```
x = ma[2];
```

OR

```
p = &ma[0];
x = *(p+2);
```

- Or we could fill the whole array with the value 77 using two approaches

```
for (int index = 0; index < 7; index++)
{
    ma[index] = 77;
}
```

```
p = &ma[0];
for (int index = 0; index < 7; index++)
{
    *p = 77; // not recommended: *(p++) = 77;
    p++;
}
```

Arrays and pointers

- It is so common to use pointers with arrays that we can refer to the address of the first element of an array in two ways:

```
int *p;
int ma[7];
```

```
p = &ma[0];
```

OR

```
p = ma;
```

Arrays and pointers

- Activity: Using only pointers, calculate the sum of an array of 100 double values and store it in a variable. The variable declarations are:

```
double values[100];
double sum;
```



POINTERS IN FUNCTION PARAMETERS

Pointers in function parameters

- C only allows the passing of parameters to functions **by value**
  - The arguments are copied to the named variables in the arguments as automatic ones
  - E.g., what value does this program display?

```c
double values[100];
double sum;

void duplicate (double value)
{
   value = 2.0 * value;
}

int main (void)
{
   double data;

   data = 5.0;
   duplicate (data);
   printf ("The value of data is %lf\n", data);
}
```

Pointers in function parameters

- **Pass by reference** in C is simulated by passing pointers by value
  - Having a pointer allows the function to modify the pointed-to data, among other things.
  - E.g., what value does this program display?

```c
void duplicate (double *value)
{
   *value = 2.0 * (*value);
}

int main (void)
{
   double data;

   data = 5.0;
   duplicate (&data);
   printf ("The value of data is %lf\n", data);
}
```

Pointers in function parameters

- E.g. this is why the *scanf()* function uses &
  - Does this function sound familiar to you? Hands up

```
int a;

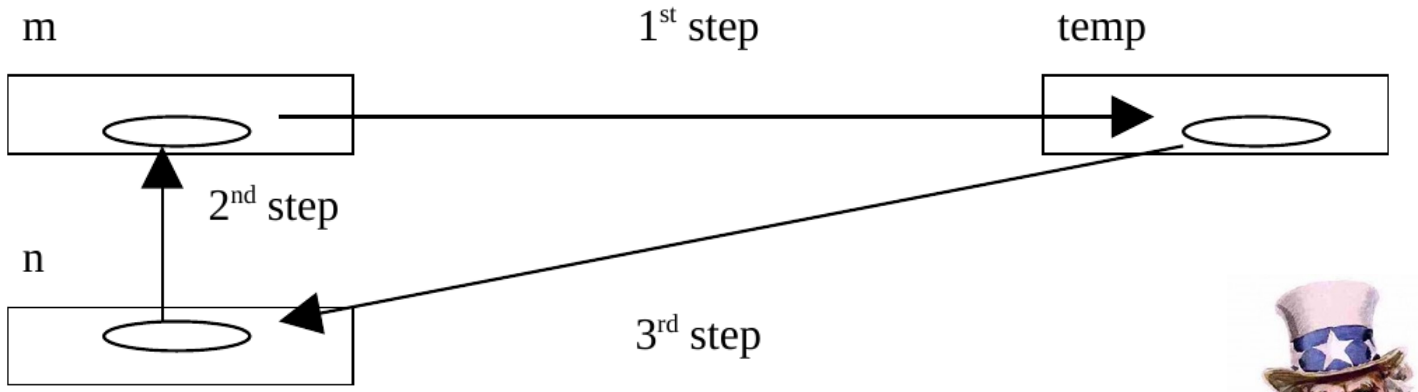
// ...
scanf("%d",&a);
```

Pointers in function parameters

- Activity: Sorting algorithms may require the capability of swapping two pieces of data. Implement the function with the prototype *void swap(double *a, double *b)* so that, when called with the addresses of two variables, it swaps their contents.
  Example of use and some tips

```
double m, n;

//...
swap (&m, &n);
```





Pointers in function parameters

- Activity: Develop a function that calculates the arithmetic mean of the values contained in arrays using the following prototype, which takes as arguments the memory address of the first value and the number of values to be processed

```c
double mean (double *p_data, int num_elements);
```

- Example of use

```c
double temperatures[N_TEMPERATURES];
double pressures[N_PRESSURES];

//...
printf("Mean temperature is %lf\n", mean(temperatures, N_TEMPERATURES));
printf("Mean pressure is %lf\n", mean(pressures, N_PRESSURES));
```



Pointers in function parameters

- NOTE: When an array is passed to a function, it decays to a pointer to its first element
- That means:
  - You cannot use *sizeof()* to know the size of the array
  - You need to pass the function the size of the data, for example the number of elements in the previous activity

- Accessing the data

$$ma[i] \equiv *(ma + i)$$

Pointers in function parameters

- As a side advantage of using pointers in functions (simulated pass by reference), data is not duplicated, achieving efficient use of memory for large data sets and faster access
- Also, data can be modified by the function, but this is an advantage in some cases and a drawback in others
- Well, if you are Java proficient, this is not new to you
- We can explicitly prohibit data modification using the data modifier **const**
- Example: this makes sense, doesn't it?

```
double mean (double const *p_data, int num_elements);
```

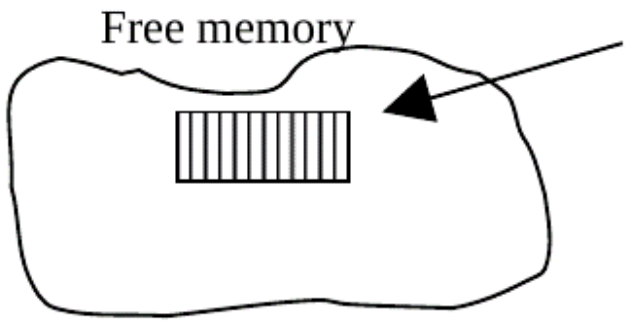
  - Anyway, it is pretty simple to circumvent this compiler "protection" … but don't do that

DYNAMIC MEMORY



Dynamic memory

- Until now, all the data structures shown are of fixed size
- The real world is different, so variable-sized structures are a must: images, text, audio, collected data … in complex structures such as lists, trees, etc.
- Dynamic memory is the mechanism used by standard C to provide this service
- It involves requesting/returning “parcels” of main memory to the C “heap”, generally

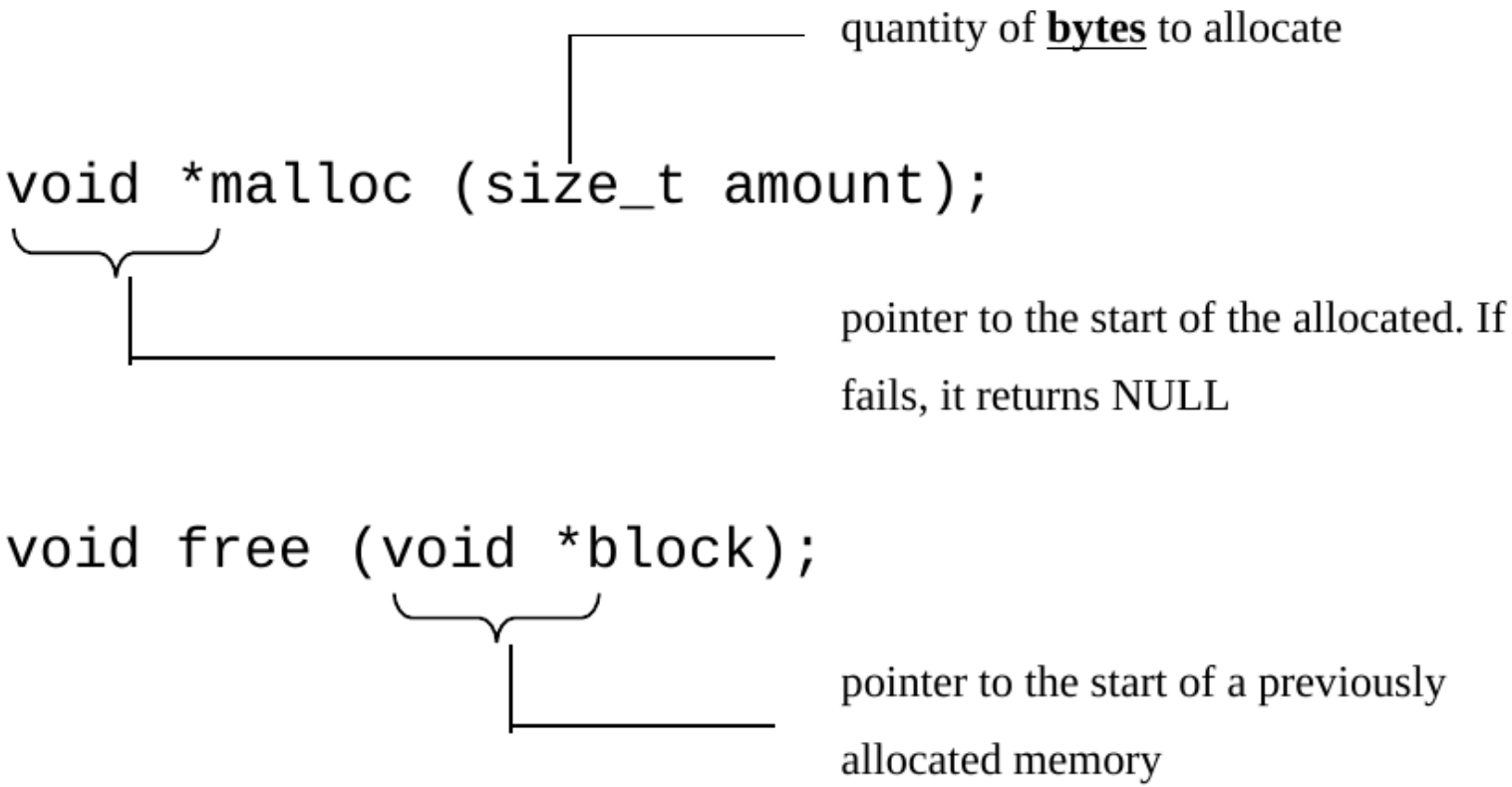


`malloc()` (memory allocation) ask C to allocate a given consecutive quantity of bytes in the “heap”

`free()` free previously requested memory (return it to the “heap”)

Dynamic memory

- The functions for this task are *malloc()*, *free()* and *realloc()*
- Access them using the header file *stdlib.h*



Dynamic memory

- Example: Allocate N doubles and store values from 1 to N in it. Note the **typecast**, optional in C, required in C++ (use it!)

typecast, convert (void *) to (double *)

required amount of bytes to store N doubles

```c
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
   int n;
   double *p, *pd;

   printf("Give me N: ");
   scanf("%d", &n);

   p = (double*) malloc (n * sizeof(double));
   if (p == NULL)
   {
    printf("No memory left!!!\n");
    exit(1);
   }

   pd = p;
   for (int index = 0; index < n; index++)
   {
      *pd = index + 1;
      pd++;
   }

   //...
   free(p);
}
```

Dynamic memory

- Activity: Write a program that prompts the user to enter a number of values, allocates memory (for doubles), prompts for the values and stores them. Using the previous *mean()* function, calculate the arithmetic mean of these values and free the allocated memory.
- To ask for numbers use:

```
scanf("%lf",&var);   // ask for a double
scanf("%d",&var);    // ask for an int
```



Dynamic memory

- Some important aspects of dynamic memory allocation in C (and other languages)
  - You can use the *realloc()* function to ask for more memory space for a given data. It could return the same pointer to the data (if extra data memory fits after the current data), a different pointer (if no contiguous data is available) or NULL

```
void *realloc(void *p_memblock, size_t size);
```

  - The *malloc()/free()* allocation algorithm fragments the memory pool, so it can fail to allocate memory if it gets too fragmented. This is especially critical on embedded systems that must run for a long period. You should consider it

    On my microcontroller-based project, I allocate memory at startup and never change it
  - In C++, the, let's say equivalent, functions for *malloc()/free()* are *new/delete*

    In fact, they also call the constructor/destructor of objects (out of scope because this subject is about C)
  - In Java, there is only new; the delete mechanism is assigned to the garbage collector

Dynamic memory

- Typical issues
  - Not checking for NULL after memory allocation
  - You lose the reference to an allocated memory block (memory leaks)
  - Using previously freed memory (use after free)
  - Freeing twice (double free)
  - Loss of previous data if *p = realloc(p, …)* returns NULL

STRUCTURES AND POINTERS

Structures and pointers

- As for other programming languages, it is typical to manage complex data using structures and dynamic memory (lists, queues, trees, …)
- When you access a structure through a pointer, you can use the **-> operator** instead of the . one

p->m ≡ (*p).m

- E.g. imagine the field *name_member* inside a structure pointed by *p_structure*

```
p_structure -> name_member
```

Structures and pointers

- Example

```c
typedef struct
{
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
} birthdate_t;

int main (void)
{
   birthdate_t data;
   birthdate_t *p_data, *p_tmp;

   // direct access to data
   data.day   = 24;
   data.month = 3;
   data.year  = 1990;

   // using pointers
   p_data = &data;
   p_data -> day   = 24;
   p_data -> month = 3;
   p_data -> year  = 1430;

   // 2 items in dynamic memory
   // NOTE: you should check this returns != NULL
   p_data = (birthdate_t *) malloc(2 *
   sizeof(birthdate_t));

   p_tmp = p_data; // preserve p_data for free()
   p_tmp -> day   = 24;
   p_tmp -> month = 3;
   p_tmp -> year  = 1525;

   p_tmp++;   //point to next element
   p_tmp -> day   = 9;
   p_tmp -> month = 11;
   p_tmp -> year  = 2026;

   free (p_data);
}
```

Structures and pointers

- Activity: Based on the previous definition of complex numbers. Implement a function to add an array of complex numbers based on the following prototype:

```c
typedef struct
{
    double real;
    double imaginary;
} complex_t;

void complex_addvector(
        const complex_t *p_data,
        int num_data,
        complex_t *result);
```

```c
// example of usage
int main(void)
{
   complex_t vector[20];
   complex_t *p;
   complex_t result;
   //...
   complex_addvector(vector, 20, &result);
   //...
   p = (complex_t *) malloc(N * sizeof(complex_t));
   //...
   complex_addvector(p,N,&result);
}
```



POINTERS: C VS JAVA

Pointers: C vs Java

Java	C
References, no arithmetic	Pointers, with arithmetic
`NullPointerException`	Undefined behaviour (a segfault if you are lucky)
`array.length`	You must pass the size
Garbage collector	`free()`

ESIGELEC

ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA