6.4 Efficiency issues

It might seem less "natural" to access the elements of an array by means of pointers, rather than the usual indexes. A good reason to opt for the pointers anyway is that they are more efficient. Let us consider the following code

```
double data[10] = {0}, a;
data[3] = 3.14;
a = data[3];
```

Each time we refer to an element of the array `data` by means of its index, the compiler needs to compute the memory position of `data[3]`. To this purpose, it must first read the value of the address in the variable `data`, compute the correct increment performing a multiplication (3×8 bytes) and finally, add the increment to `data`. If, on the opposite, we use pointers, the number of operations remains the same:

```
double data[10] = {0}, a;
*(data + 3) = 3.14;
a = *(data + 3);
```

Things are different in case of a cycle sequentially passing through all elements of an array. In the latter case it is less efficient to use indexes rather than pointers. To prove the latter statement, we consider an array of a hundred elements. We examine it in a cycle to search for elements with negative values. The Listing 6.8 contains the code cycling through the array elements.

This cycle entails, apart form the operations in the `for`, the computation of the address of `data[i]` for each control `if (data[i] < 0.)` that is performed during its execution; this amounts to executing 100 additions (`data + i`) and 100 multiplications (for each addition `i`$\times$8 bytes).

```
1 double data[100];
2 int i;
3 ... omitted code ...
4 for (i = 0; i < 100; i++) {
5   if (data[i] < 0.) {
6     printf("Negative value found %f \n", data[i]);
7   }
8 }
```

Listing 6.8 Accessing an array by means of indexes.

Instead, consider the Listing 6.9. In this case, the arithmetic of the cycle is the same. However, for none of the controls `if (*pd < 0.)` we need to

compute anything to access the current element of the array `*pd`. We also introduced an auxiliary variable `lastpd` to avoid computing 100 times the addition `data + 100` in the condition to exit the `for` cycle on line 4.

```
1 double data[100], *pd, *lastpd;
2 ... omitted code ...
3 lastpd = data + 100;
4 for (pd = data; pd < lastpd; pd++) {
5   if (*pd < 0.) {
6     printf("Negative value found %f \n", *pd);
7   }
8 }
```

Listing 6.9 Accessing an array with pointers.

In conclusion, it is preferable to perform sequential operations on an array, particularly long ones, by means of pointers to the array, rather than explicit indexes of the array. The price we pay for this greater efficiency is that the code in Listing 6.9 is less clear and, in general, the use of pointers is more prone to programming errors.

6.5 Multidimensional arrays and pointers

In Section 5.3 we studied multidimensional arrays. Let us now have a look at how to access these arrays with pointers. Recall that an array with two indexes `a[4][7]` (for instance of the `double` type) fills 4×7 contiguous memory locations. Therefore, in some way, we could consider it as a one-dimensional array of length 28. The order of the array's elements in the memory is such that the leftmost index of a multidimensional array grows more slowly, similar to how the digits of a car's odometer change.

Thus, in the memory, after the element `a[0][0]` we have `a[0][1]`, next `a[0][2]` and so on. From the operator priority viewpoint, the operator `[ ]`, appearing twice in a two-dimensional array, is evaluated as always from left to right. We can think of the array `a[4][7]` as a 4 elements array `a[4]`, whose elements are, in turn, a 7 elements array. In this viewpoint, the name `a` or also `a[0]` is a pointer to the first element of the two-dimensional array, i.e., to the first *subarray* of length 7; thus `a + 1` or `a[1]` points to the second subarray of length 7 and so on. Figure 6.1 shows this in a diagrammatic way. The expression `*(a + 1)` is equal to the content of the memory location to which `a + 1` points. This location contains the address of the second subarray. Thus `*(a + 1)` *is not an array element, but another*