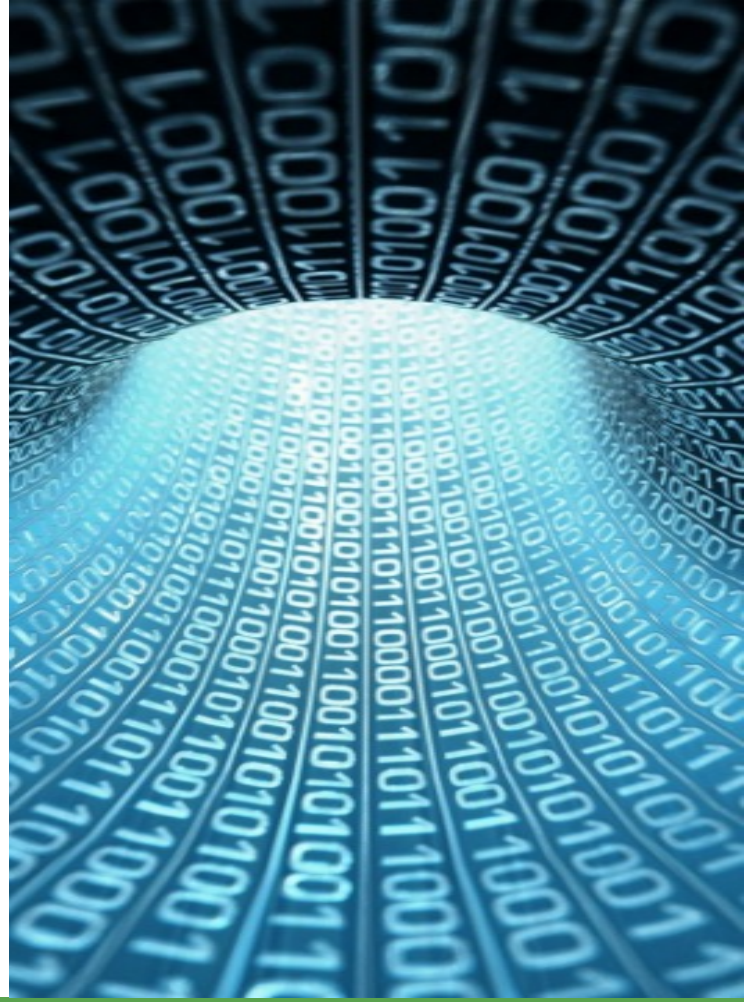
POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

Pointers-Arrays duality

POINTERS AND DYNAMIC DATA STRUCTURES: MEMORY ALLOCATION AND MODULARITY IN C LANGUAGE



Arrays and pointers

Arrays and pointers are **dual** as they allow access to data in 2 ways:

- **vectorial** with indexes and [ ]
- with **pointers** with &, * and pointers arithmetics

RULE: A **name** of a variable identifying an array formally corresponds to the **pointer** to the first element of the array:

<name array> ⇔ &<name array>[0]

Arrays and pointers

Arrays and pointers are **dual** as they allow access to data in 2 ways:

- **vectorial** with indexes and [ ]
- with **pointers** with &, * and pointers arithmetics

RULE: A **name** of a variable identifying an array formally corresponds to the **pointer** to the first element of the array:

<name array> ⇔ &<name array>[0]

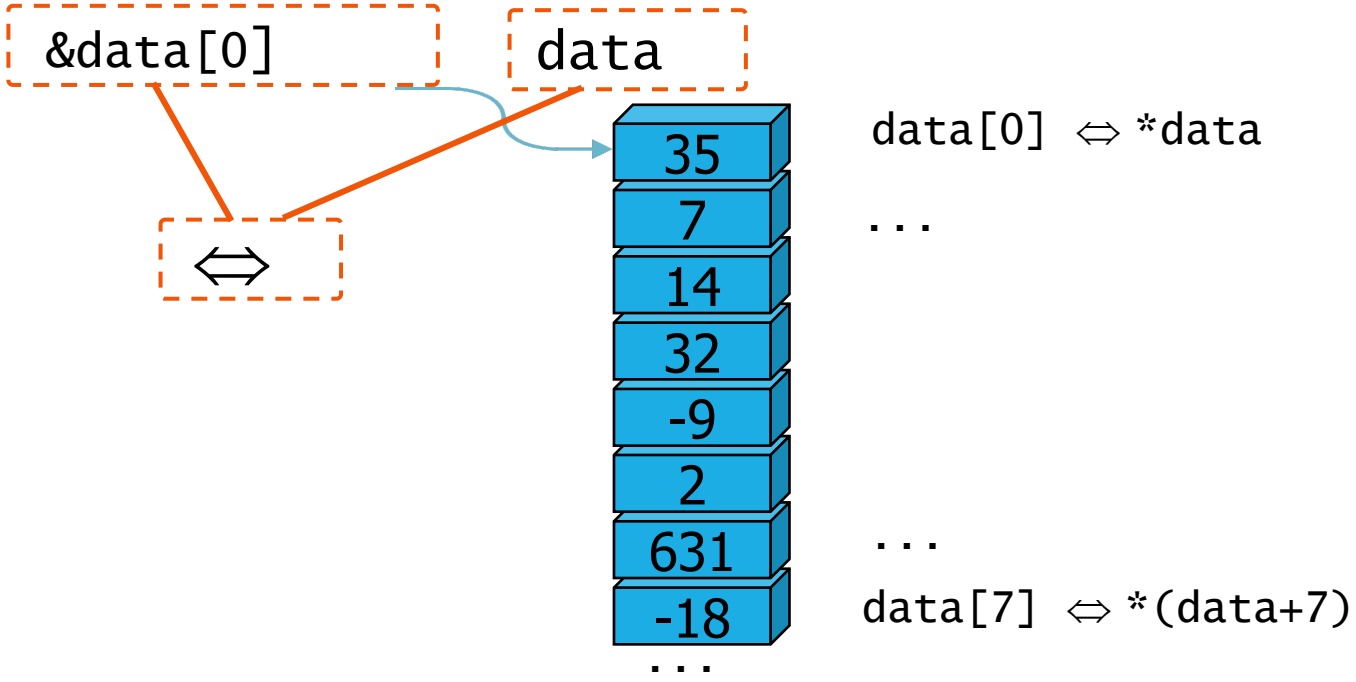
All that follows is just a consequence of this rule!

Arrays and pointers

Example: given a variable of array-type

```
int data[100];
```

- `data` ⇔ `&data[0]`
- `*data` ⇔ `data[0]`
- `*(data+i)` ⇔ `data[i]`

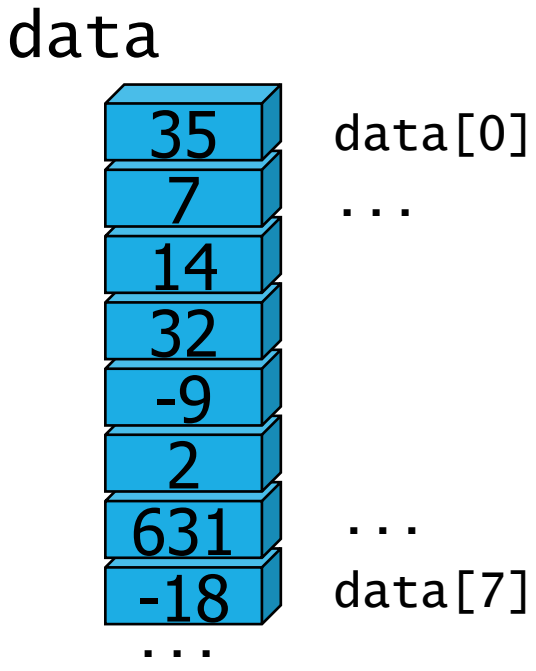

&data[0]
data
⇔
35
7
14
32
-9
2
631
-18
...
data[0] ⇔ *data
...
...
data[7] ⇔ *(data+7)

Example: read array of 100 integer values

1. As an array, with vectorial notation [ ]
   - Iterating on elements by using an index
2. With a pointer to integer `int *p`, initialized as `&data[0]`
   - Using pointers arithmetic (summation of pointer and integer index)
3. With a pointer to integer `int *p`, initialized as `&data[0]`
   - Iterating on elements directly using the pointer (which is updated at each iteration).

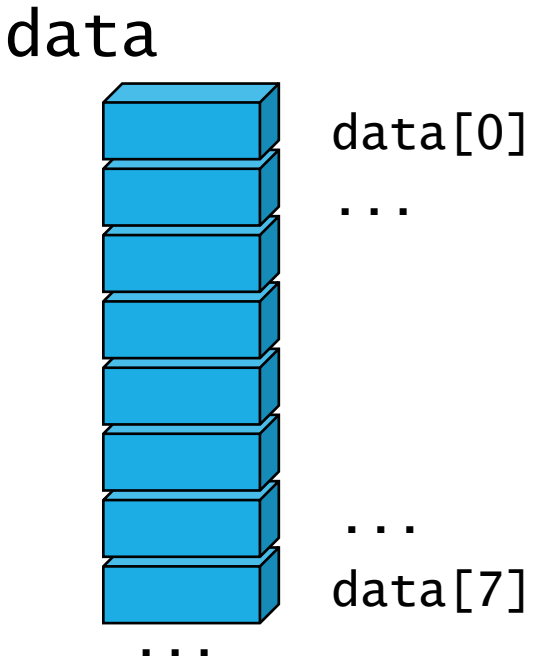
Modality 1:

```
int data[100];
...
for (i=0;i<100;i++)
  scanf("%d“,&data[i]);
```

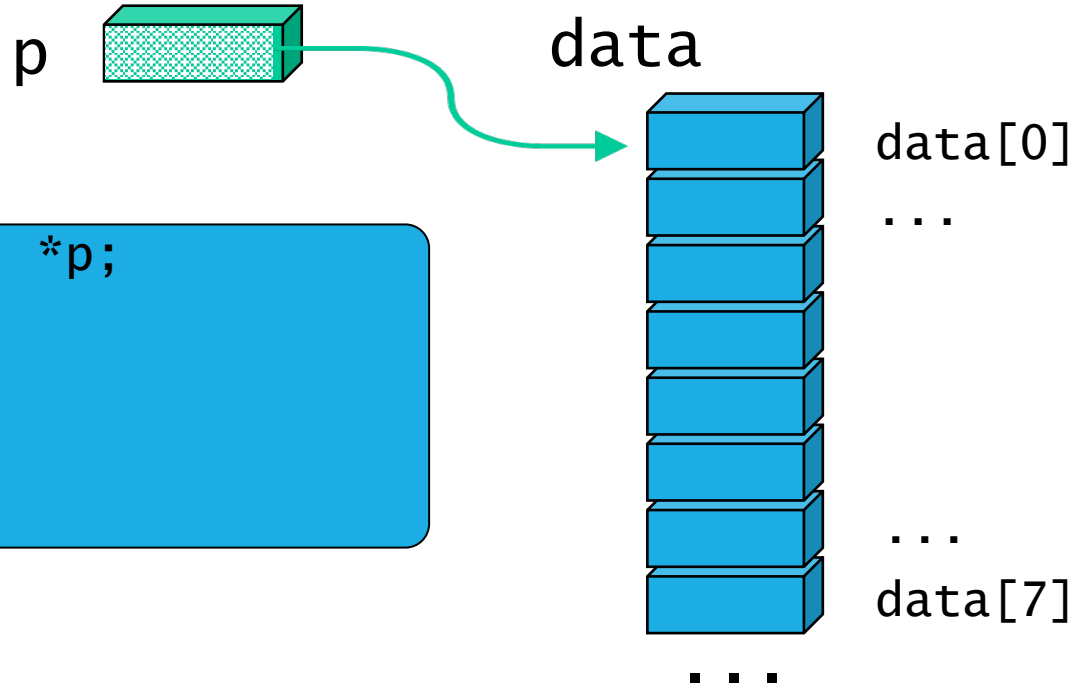


Modality 2:


```
int data[100], *p;
...
```

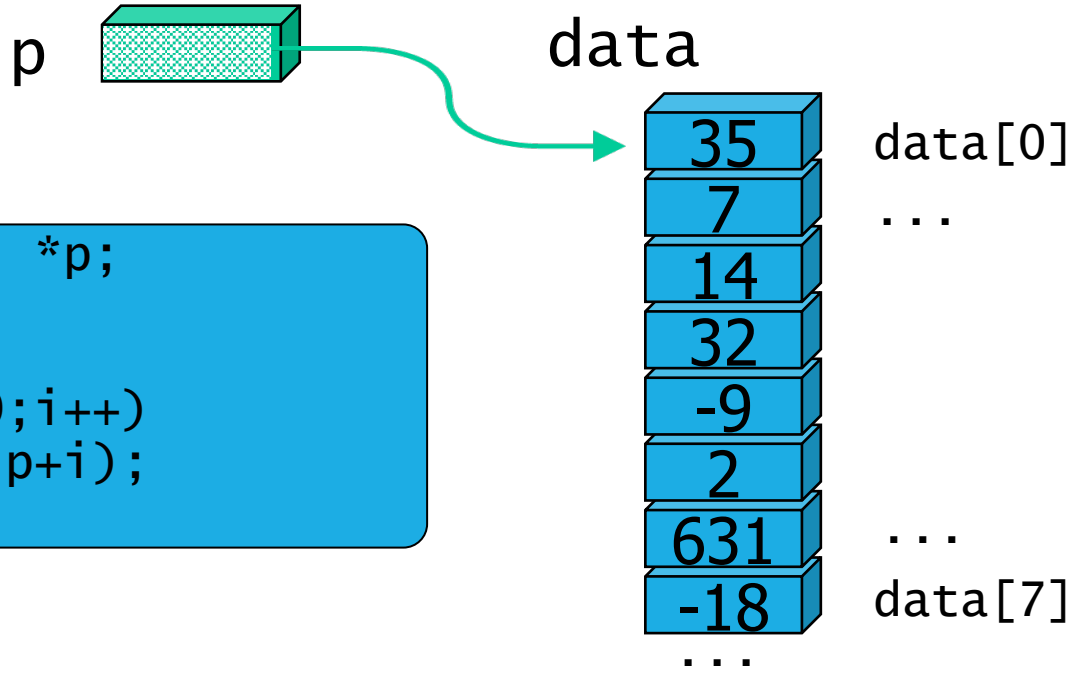


Modality 2:



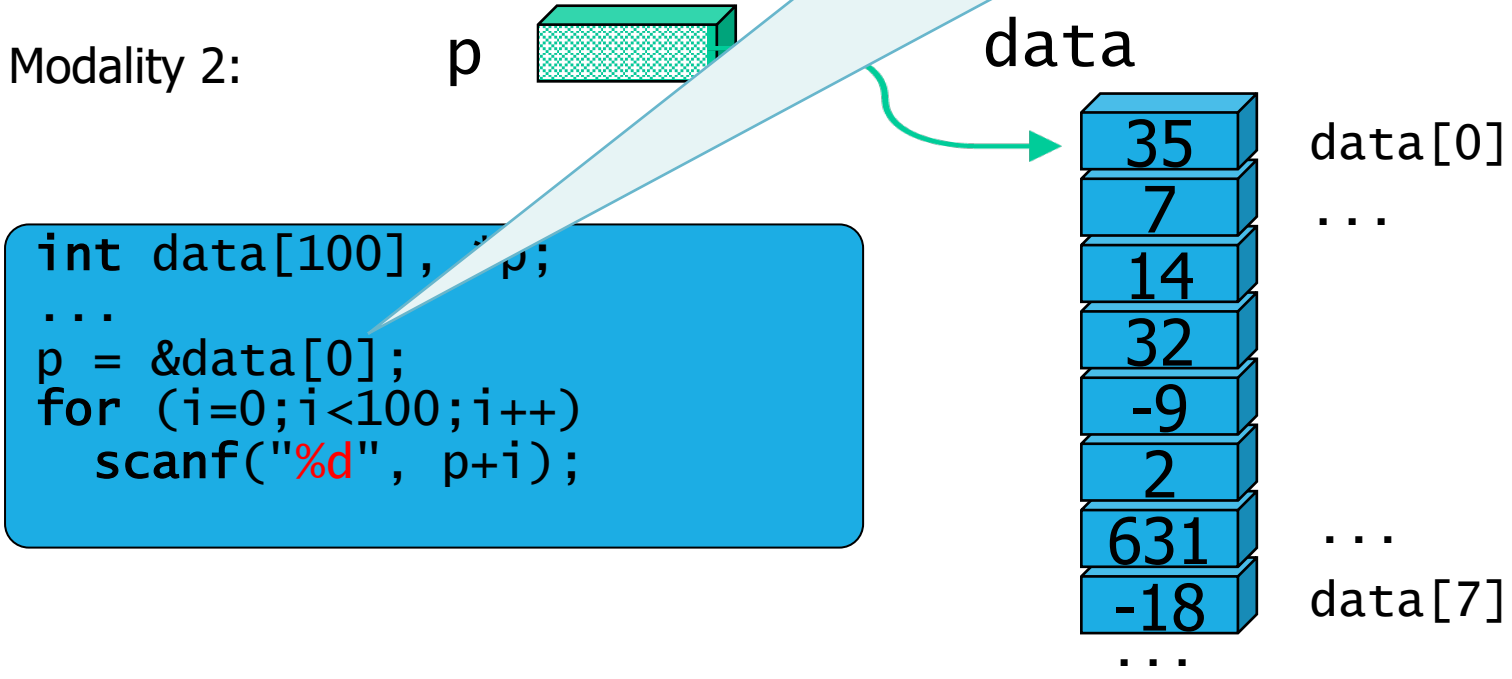

```
int data[100], *p;
...
p = &data[0];
```

Modality 2:

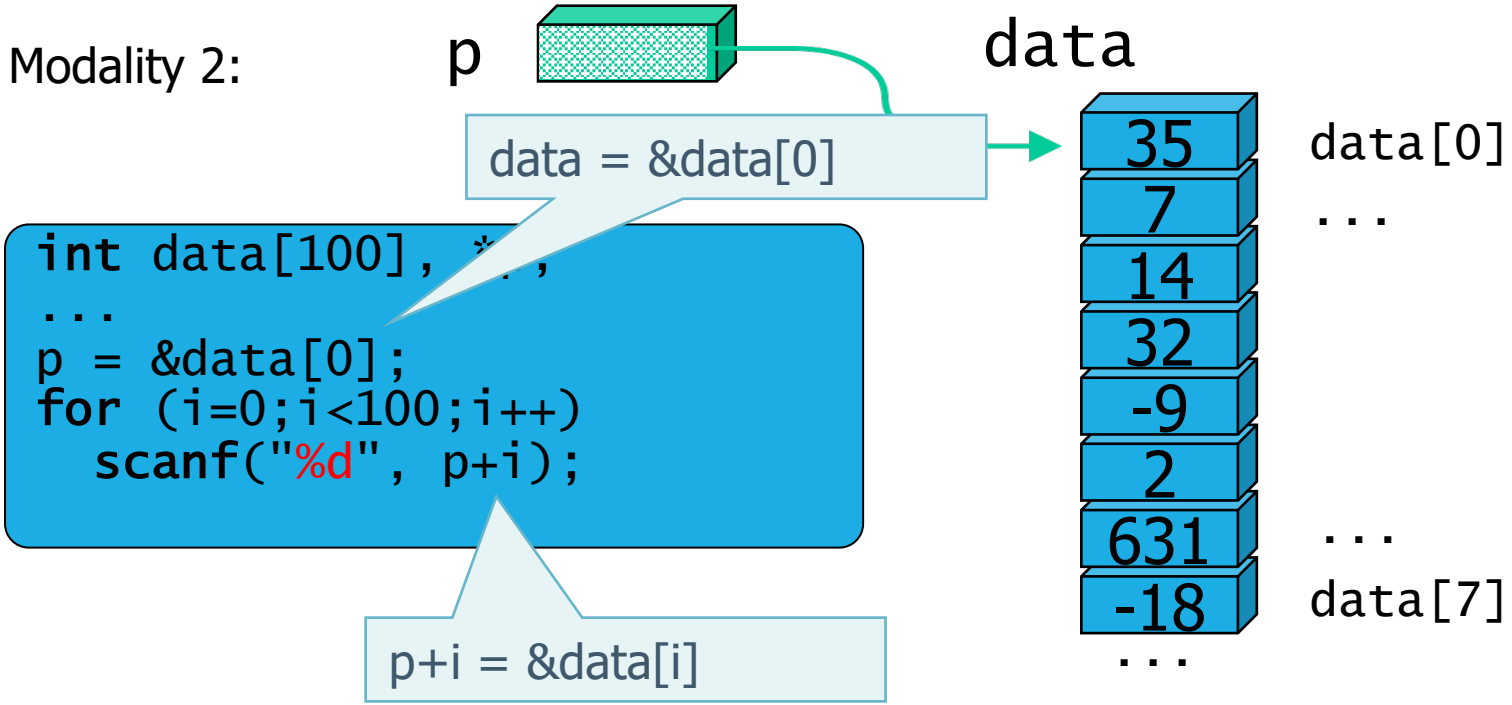



```
int data[100], *p;
...
p = &data[0];
for (i=0;i<100;i++)
  scanf("%d", p+i);
```

Modality 2:

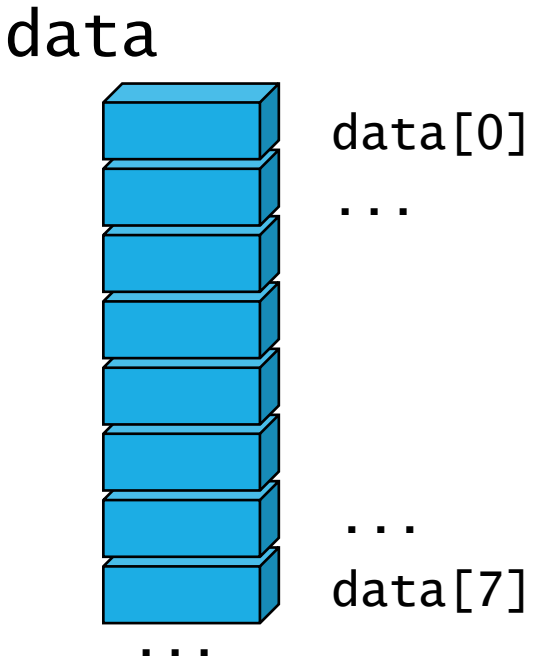

data = &data[0] (they are equivalent)
p
data
int data[100], *p;
...
p = &data[0];
for (i=0;i<100;i++)
scanf("%d", p+i);
35
7
14
32
-9
2
631
-18
...
data[0]
...
...
data[7]

Modality 2:

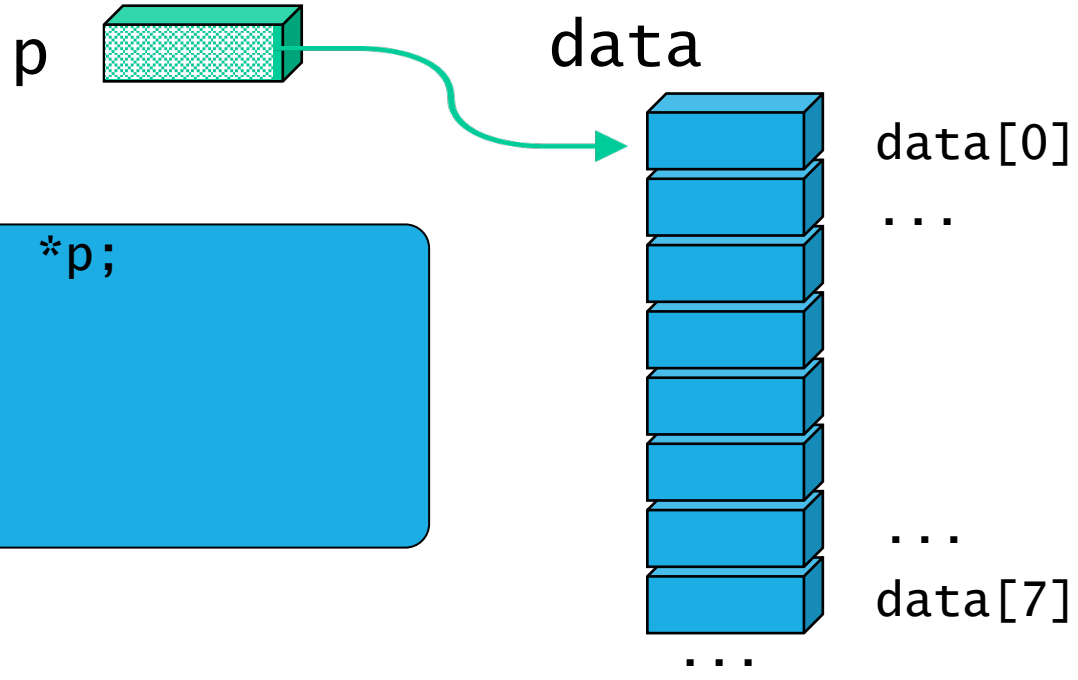

p
data
data = &data[0]
int data[100], *p;
...
p = &data[0];
for (i=0;i<100;i++)
scanf("%d", p+i);
p+i = &data[i]
35
7
14
32
-9
2
631
-18
...
data[0]
...
...
data[7]

Modality 3:


```
int data[100], *p;
...
```

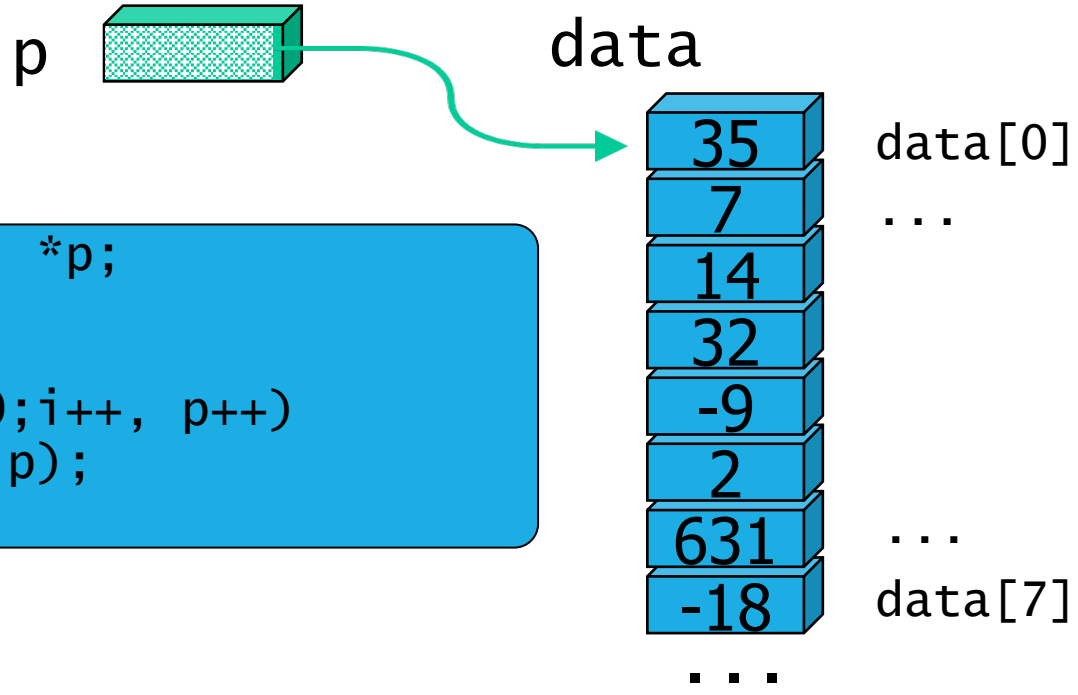


Modality 3:




```
int data[100], *p;
...
p = &data[0];
```

Modality 3:




```
int data[100], *p;
...
p = &data[0];
for (i=0;i<100;i++, p++)
  scanf("%d", p);
```

Hybrid notations are also possible:

- **pointer** to integer `int *v`, initialized as `data` (=`&data[0]`) and then used **with vectorial notation**

```
int *v = data;
for (i=0; i<100; i++)
  scanf("%d", &v[i]);
```

- **array** of integers `data` **used as a pointer**

```
for (i=0; i<100; i++)
  scanf("%d", data+i);
```

Duality limitations

- the vector name corresponds to a pointer constant, not a variable, so it cannot be incremented to scan the vector

```
for (i=0; i<100; i++, data++)
  scanf("%d", data);
```

Duality limitations

- the vector name corresponds to a pointer constant, not a variable, so it cannot be incremented to scan the vector

Pointers and subvectors

To identify a subvector between indices `l` and `r` of a given vector:

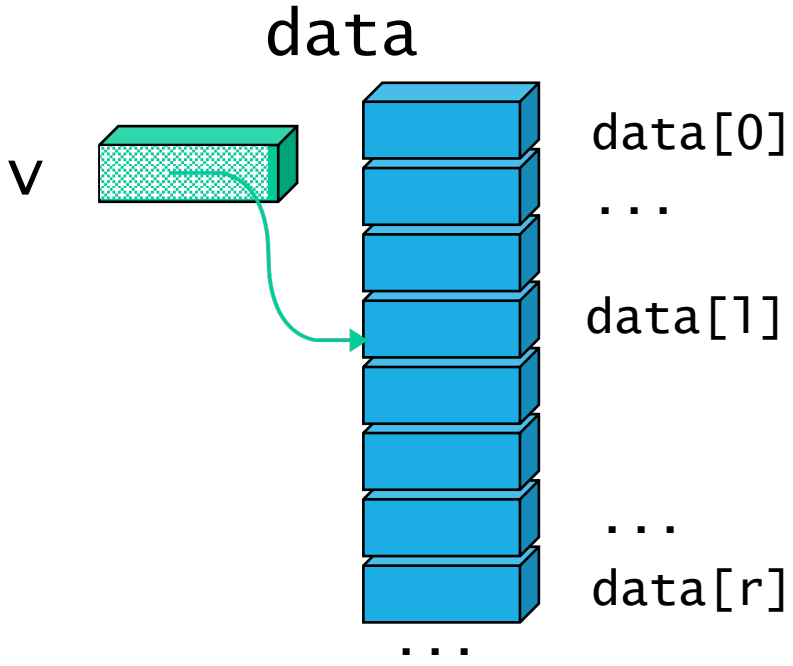
- the index i is limited between `l` and `r` (implicit identification of the sub-vector):

```
for (i=l; i<=r; i++)
  scanf("%d", &data[i]);
for (i=l; i<=r; i++)
  printf("%d", data[i]);
```

Pointers and subvectors

- a subvector is explicitly identified by a pointer to its element of index `l`:

```
int data[100], *v, i;
v = &data[l];
n = r - l + 1;
for (i=0; i<n; i++)
  scanf("%d", &v[i]);
for (i=0; i<n; i++)
  printf("%d", v[i]);
```

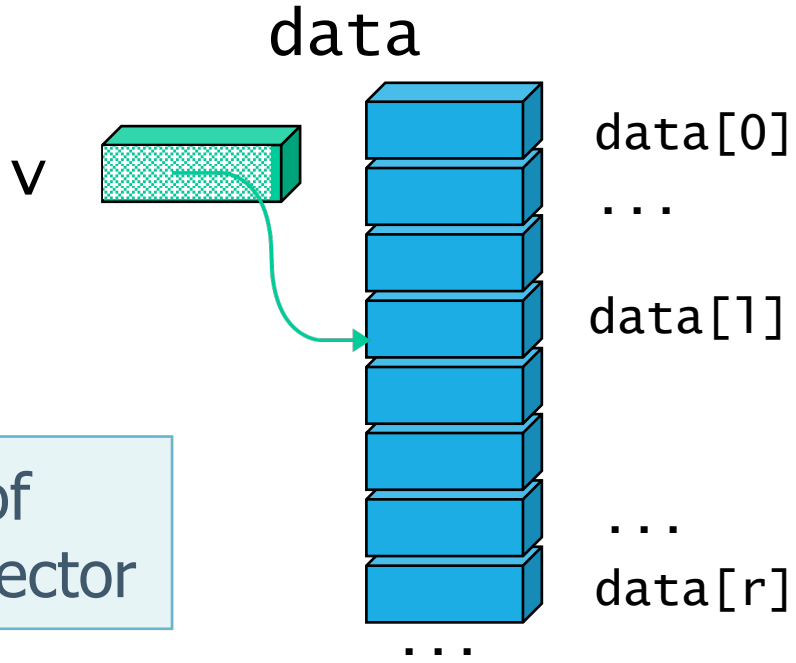


Modality 1

Pointers and subvectors

- a subvector is explicitly identified by a pointer to its element of index `l`:

```
int data[100], *v, i;
v = &data[l];
n = r - l + 1;
for (i=0; i<n; i++)
  scanf("%d", &v[i]);
for (i=0; i<n; i++)
  printf("%d", v[i]);
```



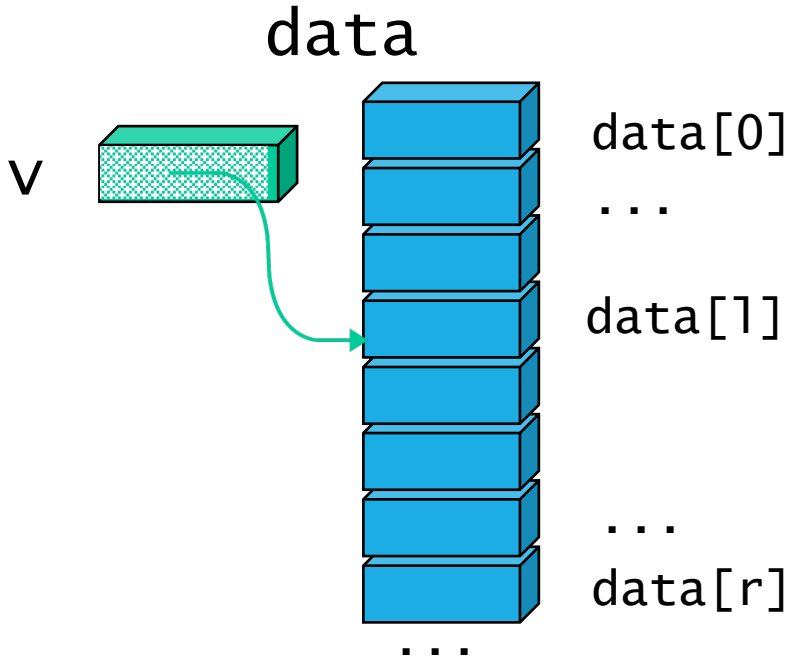
n is the number of elements of the subvector

Modality 1

Pointers and subvectors

- a subvector is explicitly identified by a pointer to its element of index `l`:

```
int data[100], *v, i;
v = &data[l];
n = r - l + 1;
for (i=0; i<n; i++)
  scanf("%d", v++);
v = data+l;
for (i=0; i<n; i++)
  printf("%d", *v++);
```

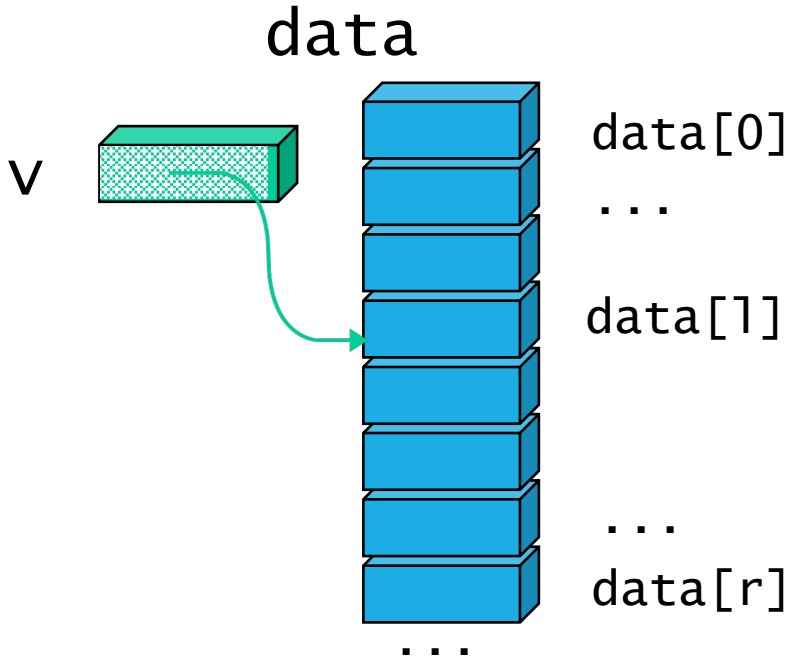


Modality 2

Pointers and subvectors

- a subvector is explicitly identified by a pointer to its element of index `l`:

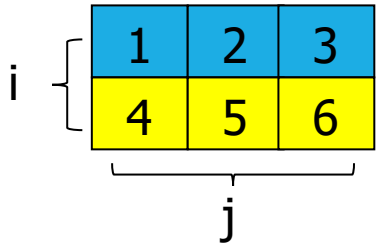
```
int data[100], *v, i;
v = data + l;
n = r - l + 1;
for (i=0; i<n; i++)
  scanf("%d", v+i);
for (i=0; i<n; i++)
  printf("%d", *(v+i));
```

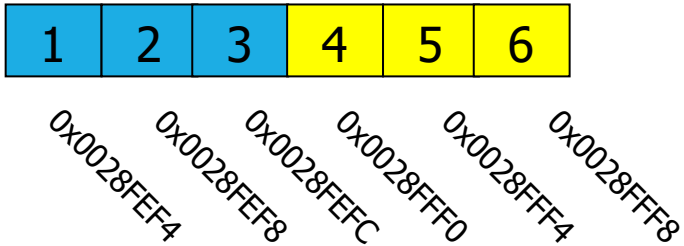


Modality 3

Matrix decomposition

- Matrixes (either bidimensional and multidimensional) can be decomposed by their rows (or sub-matrices)
- Matrixes are stored with row-major technique
  - A bidimensional matrix is stored as a vector of rows
  - All the cells of a row are stored consecutive to one another – all the rows are stored sequentially to one another

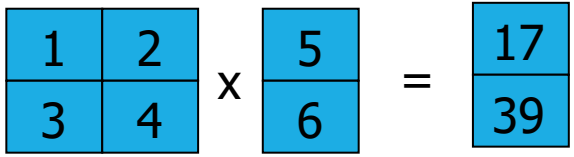




Example

Product of a matrix M of NR rows x NC columns by a vector V of NC elements.

The result is a vector of NR elements:



Modality 1

Bidimensional matrix, rows accessed by iterating on columns:

```
float M[NR][NC], V[NC], Prod[NR];
int r, c;
...
for (r=0; r<NR; r++) {
  Prod[r] = 0.0;
  for (c=0; c<NC; c++)
    Prod[r] = Prod[r] + M[r][c]*V[c];
}
```

Modality 2

Rows are identified by a pointer `row` (thanks to row-major):

```
float M[NR][NC], V[NC], Prod[NR], *row;
int r, c;
...
for (r=0; r<NR; r++) {
  Prod[r] = 0.0;
  row = M[r];
  for (c=0; c<NC; c++)
    Prod[r] = Prod[r] + row[c]*V[c];
}
```

Modality 2

Rows are identified by a pointer `row` (thanks to row-major):

```
float M[NR][NC], V[NC], Prod[NR], *row;
int r, c;
...
for (r=0; r<NR; r++) {
  Prod[r] = 0.0;
  row = M[r];
  for (c=0; c<NC; c++)
    Prod[r] = Prod[r] + row[c]*V[c];
}
```

The same as
row = &(M[r][0]);

row[c] is the same as M[r][c]

Vectors and matrixes as parameters

Vectors and matrixes passed as parameters are not generated within the function:

- By passing a name of an array to a function, we are actually passing the pointer to the first element of the array
- Only the address to the first element is passed
- Duality pointer⇔ array is:
  - Complete, for monodimensional ones (vectors)
  - Partial, for multidimensional ones (matrixes)

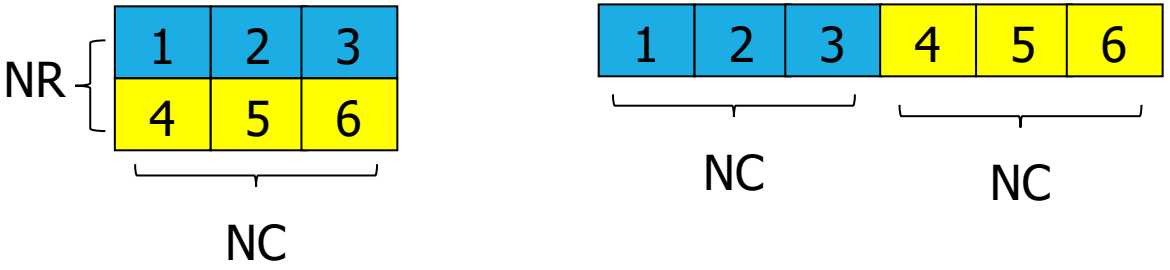
Vectors and matrixes as parameters

- To access the i-th element of a vector `vett[N]`, a function needs only the address of the first element:

  `vett[i]` is equivalent to `*(vett+i)`

- To access element [i][j] of a matrix `mat[NR][NC]`, a function needs the address of the first element AND the number of columns NC of the matrix:

  `mat[i][j]` is equivalent to `*(*mat + NC*i + j)`



Vectors and matrixes as parameters

As a consequence, when passing an array to the function, following information can be omitted in the formal parameter, as it is redundant:

- Dimension of a vector
- First (only the first!) dimension of a matrix

Examples: prototypes of library functions that have adimensional arrays as formal parameters:

```
size_t strlen(const char s[]);
int strcmp(const char s1[], const char s2[]);
char *strcpy (char dest[], const char src[]);
```

Formal parameters

Vectorial or pointer notations are interchangeable in the formal parameters

Examples: prototypes of library functions

```
size_t strlen(const char s[]);
int strcmp(const char s1[], const char s2[]);
char *strcpy (char dest[], const char src[]);
```

Examples: (real) prototypes of library functions

```
size_t strlen(const char *s);
int strcmp(const char *s1, const char *s2);
char *strcpy (char *dest, const char *src);
```

Example: arguments to `main`

A vector of pointers to char, argv, can be declared as:

- Adimensional vector of pointers

```
int main(int argc, char *argv[])
```

- Pointer to pointer (to the first element of a vector of pointers)

```
int main(int argc, char **argv)
```

Actual dimension as the parameter

Typically we pass as parameters:

- The vector without dimension
- The actual dimension (i.e., the number of elements that need to be processed), to implement a function that adapts to it

```
int read(int v[], int maxDim);
int main (void) {
  int v1[DIM1], v2[DIM2];
  int n1, n2;
  n1 = read(v1,DIM1);
  n2 = read(v2,DIM2);
  ...
}
```

```
int read(int v[], int maxDim) {
  int i, end=0;
  for (i=0; !end && i<maxDim; i++) {
    printf("v[%d] (0 to end): ", i);
    scanf("%d",&v[i]);
    if (v[i]==0) {
      end = 1;
      i--; // neglect 0
    }
  }
  return i;
}
```

Actual dimension as the parameter

Typically we pass as parameters:

- The vector without dimension
- The actual dimension (i.e., the number of elements that need to be processed), to implement a function that adapts to it

```
int read(int v[], int maxDim);
int main (void) {
  int v1[DIM1], v2[DIM2];
  int n1, n2;
  n1 = read(v1,DIM1);
  n2 = read(v2,DIM2);
  ...
}
```

```
int read(int v[], int maxDim) {
  int i, end=0;
  for (i=0; !end &&  i<maxDim; i++) {
    printf("v[%d] (  o end): ", i);
    scanf("%d",&v[i]);
    if (v[i]==0) {
      end = 1;
      i--; // neglect 0
    }
  }
  return i;
```

I can call the same function on arrays of different dimensions

It is the same as:
int read(int *v, int maxDim)

Formal-actual parameters correspondence

Formal parameter vector - actual pointer

- Allows to generate an array (for a function) starting from a sub-vector or from a pointer (to adjacent memory)

Example: sort a vector by groups

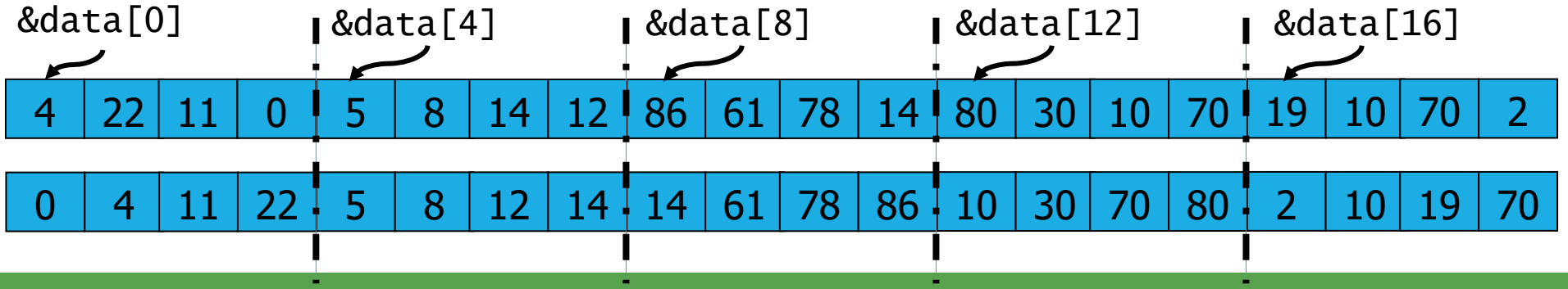
```
void sortInt(int v[], int n);
...
int data[20];
...
for (i=0;i<20;i+=4)
  sortInt(&data[i],4);
```

Formal parameter vector - actual pointer

Example: sort a vector by groups

```
void sortInt(int v[], int n);
...
int data[20];
...
for (i=0;i<20;i+=4)
  sortInt(&data[i],4);
```

Sorting applied to sub-vectors of 4 elements



Formal parameter pointer - actual vector

- To a formal parameter that is a pointer can correspond an actual parameter that is a vector
  - The pointer can then internally handled as a vector

```
int read(int *v, int maxDim);
...
int dim, data[100];
...
dim = read(data, 100);
```

Pointers as strings

STRINGS MANIPULATED AS VECTOR OR BY USING POINTERS

Pointers and strings

- There is not a string type in C
- A string is (formally) an array of char elements, terminated by a ’\0’
- Strings can be manipulated:
  - As vectors
  - Using pointers and pointers arithmetic

Strlen (version 0)

Function equivalent to a `strlen`, with strings as a vector :

- Count characters until '\0'

```
int strlen0(char s[]){
  int cnt=0;
  while (s[cnt]!='\0')
    cnt++;
  return cnt;
}
```

Strlen (version 0)

Function equivalent to a `strlen,` with strings as a vector :

- Count characters until '\0'

```
int strlen0(          []){
  int cnt=0;
  whi
    c
  ret
}
```

ATTENTION: it is not the real library function `strlen,` just an equivalent implementation

Strlen (version 0)

Function equivalent to a `strlen`, with strings as a vector :

- Count characters until '\0'

```
int strlen0(char s[]){
  int cnt=0;
  while (s[cnt]!='\0')
    cnt++;
  return cnt;
}
```

`cnt` is both an index and a counter

Strlen (version 1)

Function equivalent to `strlen` with string as a pointer:

- Scans string with a pointer p
- Counts with integer counter `cnt`

```
int strlen1(char s[]){
  int cnt=0;
  char *p=&s[0];
  while (*p !='\0'){
    cnt++;
    p++;
  }
  return cnt;
}
```

`cnt` serves as a counter, strings scanning done with pointer p

Strlen (version 2)

Function equivalent to `strlen` with string as a pointer:

- Scans the string directly with the pointer `s`
- Counts with integer counter `cnt`

```
int strlen2(char *s){
  int cnt=0;
  while (*s++ != '\0')
    cnt++;
  return cnt;
}
```

`cnt` serves as a counter, strings scanning done with pointer `s`

Strlen (version 3)

Function equivalent to `strlen` with string as a pointer:

- Scans the string with a pointer p
- It does not count, uses pointer arithmetic (difference of pointers)

```
int strlen3(char *s){
  char *p = s;
  while (*p != '\0')
    p++;
  return p-s;
}
```

We do not need explicit counter p scans the string.

Example: `strcmp`

- Given two strings, compare their content and return:
  - 0 if strings are equal
  - <0 if first string precedes the second
  - >0 if first string follows the second

  If the strings are different, the function returns the difference of the ASCII characters of the first pair of different characters

  - strcmp("Hello","Help") -> 'l'-'p' = -4
  - strcmp("Hello","Hello") -> 0

Strcmp (version 0)

- Strings as vectors
- Strategy:
  - Iteration to compare characters until the first different pair, or until string termination
  - Return difference between different ASCII char (or difference between terminators)

```
int strcmp0(char s0[], char s1[]){
  int i=0;
  while (s0[i]==s1[i] && s0[i]!='\0')
    i++;
  return (s0[i]-s1[i]);
}
```

Strcmp (version 1)

- Strings as pointers
- Same strategy, but iteration handled by pointers

```
int strcmp1(char *s0, char *s1) {
  while ((*s0==*s1) && (*s0!='\0')){
        s0++;
        s1++;
  }
  return (*s0-*s1);
}
```

Example: strncmp (version 0)

- The function is limited to the first n characters of the two strings
- Just like `strcmp0`, but with comparison limited to n characters

```
int strncmp0(char s0[], char s1[], int n){
  int i=0;
  while (s0[i]==s1[i] && s0[i]!='\0')
    if (i<n)
      i++;
    else
      return 0;
  return (s0[i]-s1[i]);
}
```

Example: strncmp (version 1)

- Same strategy as before, but with pointers…

```
int strncmp0(char *s0, char *s1, int n){
  int i=0;
  while ((*s0==*s1) && (*s0!='\0')){
    if (i<n)
      {s0++; s1++;}
    else
      return 0;
  return (*s0-*s1);
}
```

Example: strstr

Given two strings, search for the second string inside the first one and return a pointer:

- NULL, if the search is not successful
- To the first character of the first occurrence of the sub-string

```
char *strstr0(char s[], char c[]){
  int i, lun_s, lun_c;
  lun_s=strlen(s);
  lun_c=strlen(c);
  for (i=0; i<=lun_s-lun_c; i++)
    if (strncmp(&s[i],c,lun_c)==0)
      return (&s[i]);
  return (NULL);
}
```

Example: `strstr`

Given two strings, search for the second string inside the first one and return a pointer:

- `NULL`, if the search is not successful
- To the first character of the first occurrence c

```
char *strstr0(char s[], char c[]){
  int i, lun_s, lun_c;
  lun_s=strlen(s);
  lun_c=strlen(c);
  for (i=0; i<=lun_s-lun_c; i++)
    if (strncmp(&s[i],c,lun_c)==0)
      return (&s[i]);
  return (NULL);
}
```

Iteration that, for the i-th character of the first string s, determines whether it is the starting point of the sub-string that we are looking for (using `strncmp`)

Example: input with `sscanf`

- Acquire from a line of text (`stdin`) a string (of no more than MAX characters), containing an unknown number of integers separated by space
- Compute and print the arithmetic average of the acquired values
- Strategy:
  - We cannot use formatted input (%d), as we do not know the number of fields, and %d does not differentiale spaces and return characters.
  - Input in two steps:
    - `gets` (or `fgets`) to acquire a line of text, as a string
    - Acquisition of integers from the line using `sscanf`.

Example: input with `sscanf`

- First attempt: WRONG!!!
  - `sscanf` "restarts" each time from the beginning of the line (it does not update automatically like fscanf would do while reading from a file).
  - Hence, only the first value is acquired

```
// declaration of variables
...
fgets(line,MAX,stdin);
while (sscanf(line, "%d", &x)>0) {
  sum += x;
  cnt++;
}
printf("The average is: %f\n", sum/cnt);
```

Example: input with sscanf

- First attempt: WRONG!!!
  - sscanf "restarts" each time from the beginning of the line (it does not update automatically like fscanf would do while reading from a file).
  - Hence, only the first value is acquired

```
// declaration of variables
...
fgets(line,MAX,stdin);
while (sscanf(line, "%d", &x)>0) {
  sum += x;
  cnt++;
}
printf("The average is: %f\n", sum/cnt);
```

Correct strategy

- We cannot use formatted input (%d), as we do not know the number of fields, and %d does not differentiale spaces and return characters.
- Input in two steps:
  - `gets` (or `fgets`) to acquire a line of text, as a string
  - Acquisition of integers from the line using `sscanf`
- PROBLEM: how to iterate `sscanf(line, "%d", …)` so that we can start from where we were left at the previous iteration?
- SOLUTION:
  - format %n (it says how many characters were read)
  - pointer to identify a sub-string

Solution

```
...
int i, x, cnt = 0;
float sum = 0.0;
char line[MAX], *s;
...
fgets(line,MAX,stdin);
s=line;
while (sscanf(s, "%d%n", &x, &i)>0) {
  s = s+i; // or s = &s[i];
  sum += x;
  cnt++;
}
printf("The average is %f\n", sum/cnt);
```

Solution

```
...
int i, x, cnt = 0;
float sum = 0.0;
char line[MAX], *s;
...
fgets(line,MAX,stdin);
s=line;
while (sscanf(s, "%d%n", &x, &i)>0) {
  s = s+i; // or s = &s[i];
  sum += x;
  cnt++;
}
printf("The average is %f\n", sum/cnt);
```

- `s` points to the start of the "portion" of the string we are interested in
- `i` "says" (thanks to %n) how many characters were read until then
- `i` is used to update `s`

Vectors of pointers

MATRIXES GENERATED AS VECTORS OF POINTERS (IN SUB-MATRXES)

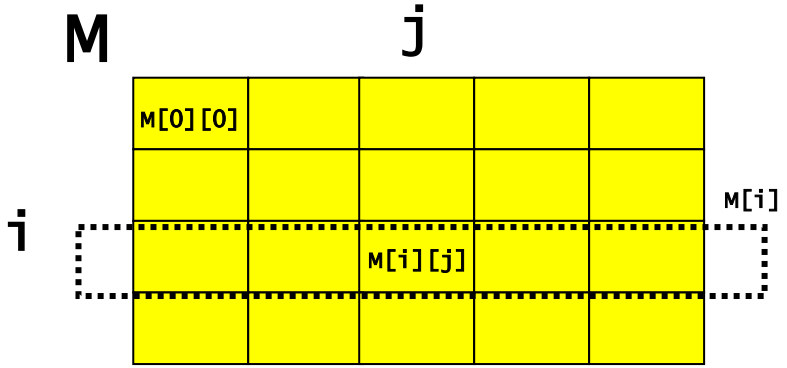
Vectors of pointers

- As a pointer is a datum
  - There could also be vectors of pointers
- As a pointer can correspond to a vector
  - Then, a vector of pointers can correspond to a vector of vectors (i.e., to a matrix)
- ATTENTION!
  - There exists a duality, but a matrix that is realized as a vector of pointers is different from a matrix that is declared with vectorial notation (with [])

Matrix as a vector of rows

Example: matrix as a vector of rows (NO POINTERS!)

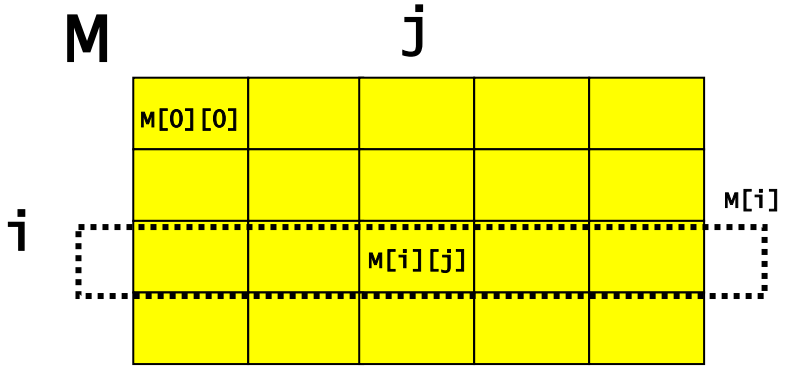
```
#define NR 4
#define NC 5
float M[NR][NC];
...
```



Matrix as a vector of rows

Example: matrix as a vector of rows

```
#define NR 4
#define NC 5
float M[NR][NC];
...
```



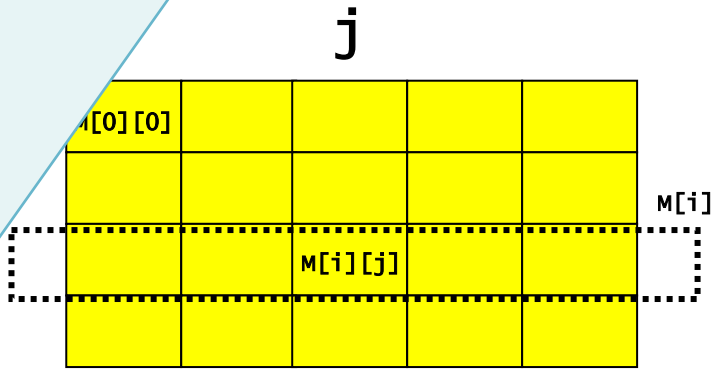

```
printf("dim. matr.: %d\n", sizeof(M));
printf("dim. row: %d\n", sizeof(M[0]));
printf("dim. elem.: %d\n", sizeof(M[0][0]));
```

Matrix as a vector of rows

Exampl

80 = NR x NC x sizeof(float)

```
#define NR 4
#define NC 5
float M[NR][NC];
...
```



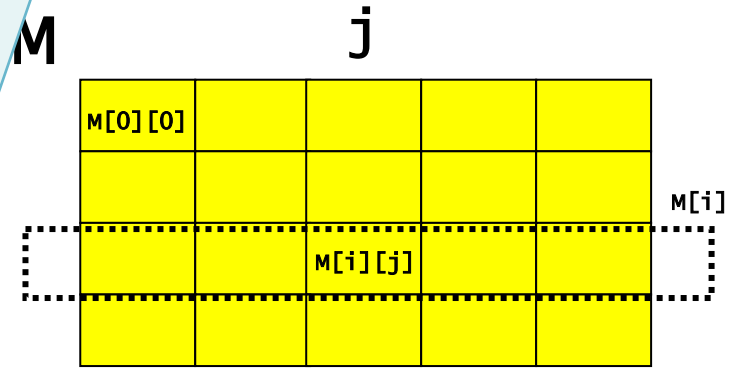

```
printf("dim. matr.: %d\n", sizeof(M));
printf("dim. row: %d\n", sizeof(M[0]));
printf("dim. elem.: %d\n", sizeof(M[0][0]));
```

Matrix as a vector of rows

Example: matrix as a vec

20 = NC x sizeof(float)

```
#define NR 4
#define NC 5
float M[NR][NC];
...
```



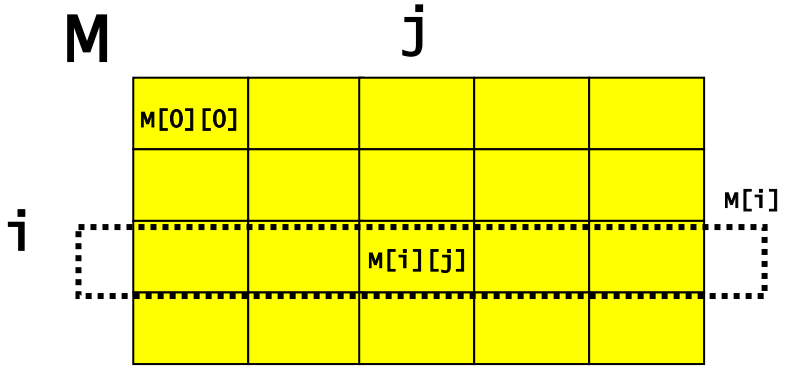

```
printf("dim. matr.: %d\n", sizeof(M));
printf("dim. row: %d\n", sizeof(M[0]));
printf("dim. elem.: %d\n", sizeof(M[0][0]));
```

Matrix as a vector of rows

Example: matrix as a vec

```
4 = sizeof(float)
```

```
#define NR 4
#define NC 5
float M[NR][NC];
...
```

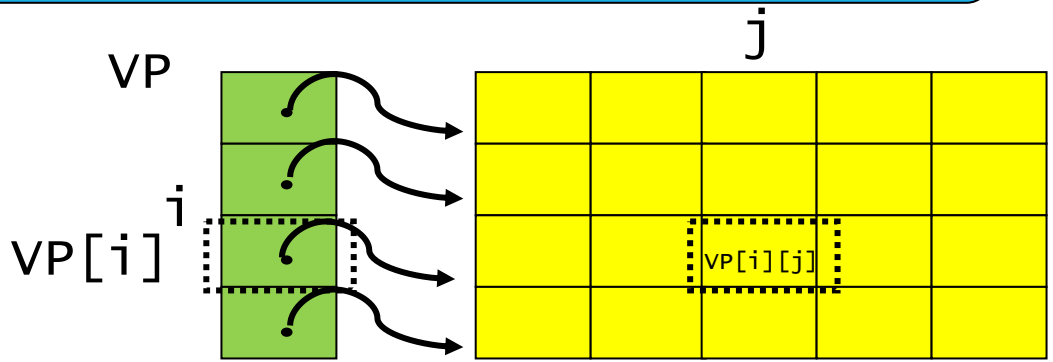



```
printf("dim. matr.: %d\n", sizeof(M));
printf("dim. row: %d\n", sizeof(M[0]));
printf("dim. elem.: %d\n", sizeof(M[0][0]));
```

Matrix as vector of pointers (per rows)

Matrix as vector of pointers to (the initial cells of) vectors

```
#define NR 4
#define NC 5
float R0[NC],R1[NC],R2[NC],R3[NC];
float *VP[NR] = {R0,R1,R2,R3};
...
```

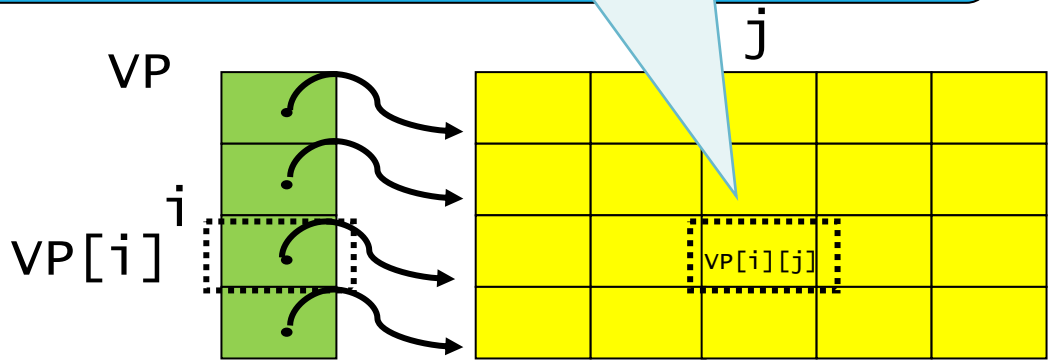


Matrix as vector of pointers (per rows)

Matrix as vector of pointer

```
#define NR 4
#define NC 5
float R0[NC],R1[NC],R2[NC],R3[
float *VP[NR] = {R0,R1,R2,R3};
...
```

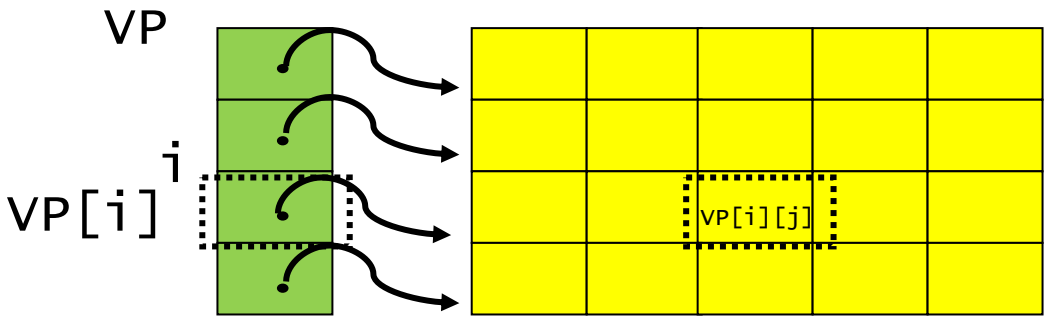
Matricial notation for
VP[i][j] ⇔(VP[i])[j]
(even though VP is a vector)



Dimensions:

- 32 byte for `VP` (vector of 4 pointers)
- 8 byte for `VP[i]` (pointer)
- 4 byte for `VP[i][j]` (float)

We assume that the size of a pointer is 8 byte (64 bit processor)



Matrix as a vector of pointers (to rows)

Dimensions:

- 32 byte for `VP` (vector of 4 pointers)
- 8 byte for `VP[i]` (pointer)
- 4 byte for `VP[i][j]` (float)

Same notation but different results:
`M` and `VP` are not the same thing

```
printf("dim. matr.: %d\n", sizeof(VP));
printf("dim. row: %d\n", sizeof(VP[0]));
printf("dim. elem.: %d\n", sizeof([0][0]));
```

Vectors of vectors of variable dimension

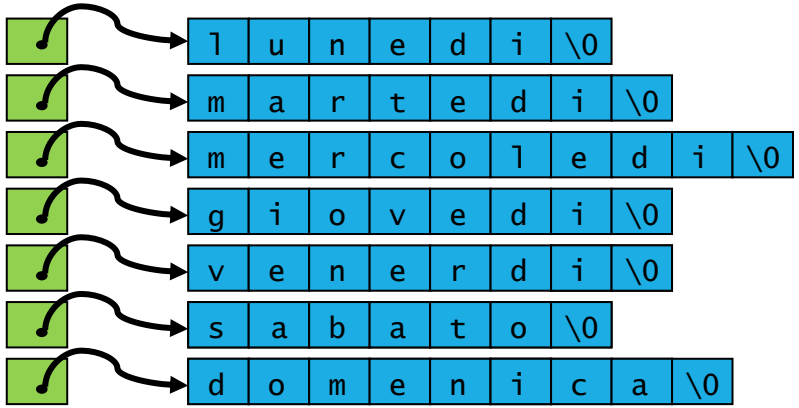
Thanks to vectors of pointers (vectors of vectors), with a matricial notation we can obtain matrixes with rows of variable dimension

- This is a possibility that is often exploited in vectors of strings, that can also be accessed as matrixes of characters

Example: vectors with the names of days of the week in italian. Print the i-th character of each name, in case it exists (i acquired from keyboard)

- solution 1: matrix di characters
- solution 2: vector of strings (pointers to char)

l	u	n	e	d	i	\0			
m	a	r	t	e	d	i	\0		
m	e	r	c	o	l	e	d	i	\0
g	i	o	v	e	d	i	\0		
v	e	n	e	r	d	i	\0		
s	a	b	a	t	o	\0			
d	o	m	e	n	i	c	a	\0	



solution 1: matrix di characters

```
void main (void) {
  int i,d;
  char days[7][10]={"lunedi","martedi",
                    "mercoledi","giovedi",
                     "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

solution 1: matrix di characters

```
void main (void) {
  int i,d;
  char days[7][10]={"lunedi","martedi",
                    "mercoledi","giovedi",
                     "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

Initialization with string constants

solution 1: matrix di characters

```
void main (void) {
  int i,d;
  char days[7][10]={"lunedi","martedi",
                    "mercoledi","giovedi",
                     "venerdi","sabato"  ica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

Generic row of the matrix identified by one index

solution 1: matrix di characters

```
void main (void) {
  int i,d;
  char days[7][10]={"lunedi","martedi",
                    "mercoledi","giovedi",
                     "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

Individual character identified with two indexes

solution 2: vector of pointers to char

```
void main (void) {
  int i,d;
  char *days[7]={"lunedi","martedi",
                    "mercoledi","giovedi",
                     "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

solution 2: vector of pointers to char

```
void main (void) {
  int i,d;
  char *days[7]={"lunedi","martedi",
                 "mercoledi","giovedi",
                  "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

vector of pointers to char

solution 2: vector of pointers to char

```
void main (void) {
  int i,d;
  char *days[7]={"lunedi","martedi",
                 "mercoledi","giovedi",
                  "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

Initialization with pointers to constant strings

solution 2: vector of pointers to char

```
void main (void) {
  int i,d;
  char *days[7]={"lunedi","martedi",
                 "mercoledi","giovedi",
                  "venerdi","sabato","domenica"};
  printf(”which character(1-6)? ");
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

days[d]:d-th string

solution 2: vector of pointers to char

```
void main (void) {
  int i,d;
  char *days[7]={"lunedi","martedi",
                 "merc
                  "vene
  printf(”which character(1
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

days[d][i-1]: (i-1)-th character of the d-th string

solution 2: vector of pointers to char

```
void main (void) {
  int i,d;
  char *days[7]={"lunedi","martedi",
                 "merc
                  "vene
  printf(”which character(1
  scanf("%d",&i);
  for (d=0; d<7; d++)
    if (i<strlen(days[d]))
      printf("%c ", days[d][i-1]);
    else printf("_ ");
  printf("\n");
}
```

days[d][i-1]: vector of strings used like a matrix of characters

Vector of strings

- A vector of strings can be realized as:
  - Matrix of characters: bidimensional array (rows, columns). The rows have all the same dimension (overized based on the longest string that needs to be stored)
  - Vector of pointers to char: each element of the vector points to a different string. The strings can have different dimensions
- It is possible to use the matricial notation with both methods

Example: sorting of strings

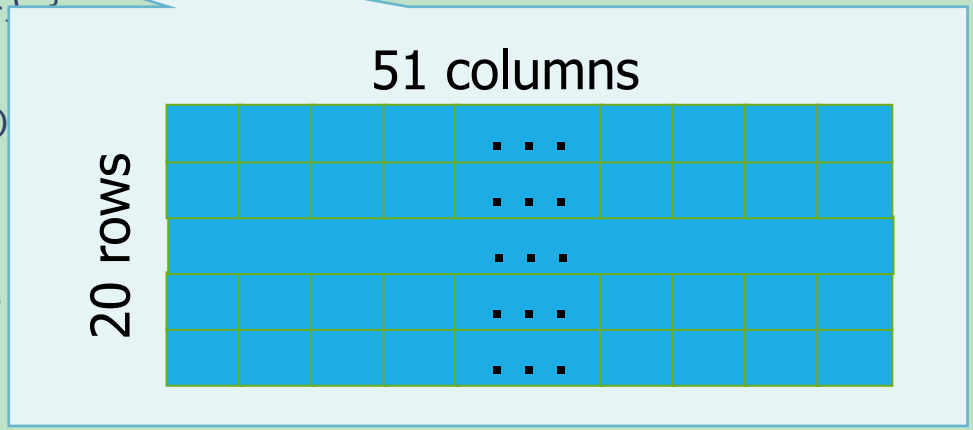
- Read strings from keyboard:
  - At most 20
  - Each string is long at most 50 characters
  - The total length of the strings is <= 500
  - The input terminates with a empty string
- Sort the strings in ascending order (based on `strcmp`)
- Visualize the sorted strings on the screen

Solution 1: matrix of characters

```
void main (void){
  int i,ns;
  char m[20][51]; // 51 columns to take \0 into account
  printf(”Insert the strings:\n");
  for (ns=0; ns<20; ns++) {
    gets(m[ns]);
    if (strlen(m[ns])==0) break;
  }
  sortMatrix(m,ns);
  printf(”sorted strings:\n");
  for (i=0; i<ns; i++)
     printf("%s\n", m[i]);
}
```

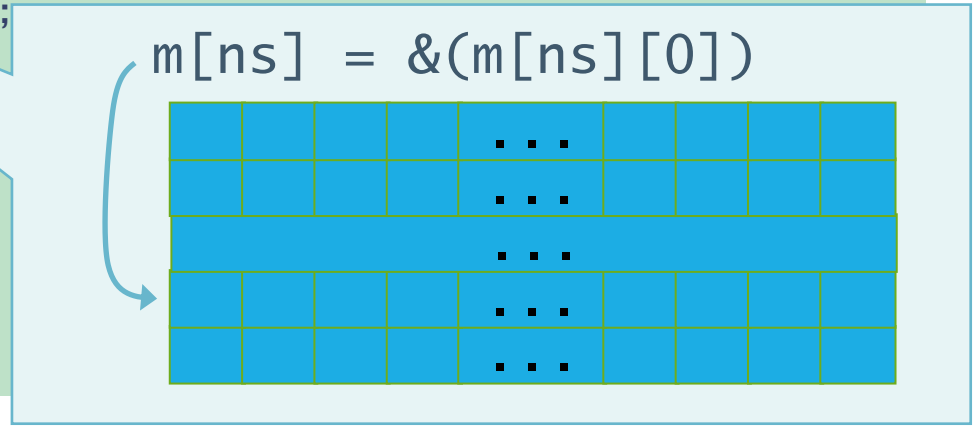
Solution 1: matrix of characters

```
void main (void){
  int i,ns;
  char m[20][51]; // 51 columns to take \0 into account
  printf("Insert the st  gs:\n");
  for (ns=0; ns<20; ns++) {
    gets(m[ns]);
    if (strlen(m[ns])==0)
  }
  sortMatrix(m,ns);
  printf("sorted strings
  for (i=0; i<ns; i++)
  printf("%s\n", m[i]);
}
```



Solution 1: matrix of characters

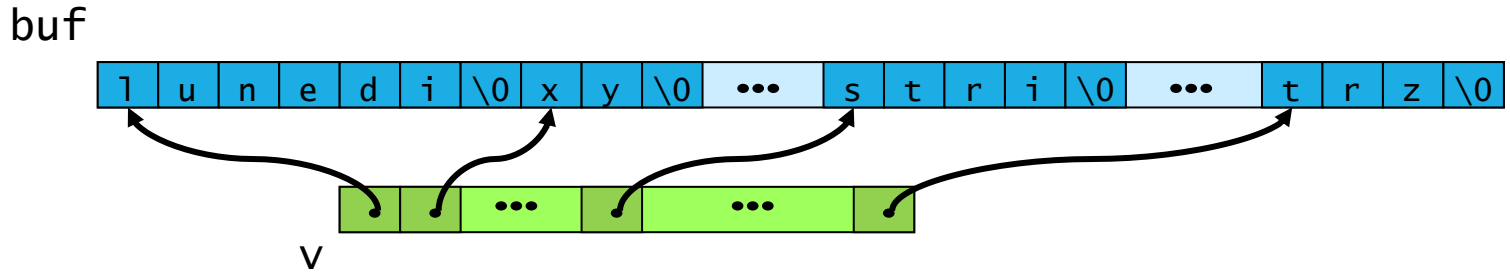
```
void main (void){
  int i,ns;
  char m[20][51]; // 51 columns to take \0 into account
  printf(”Insert the strings:\n");
  for (ns=0; ns<20; ns++) {
    gets(m[ns]);
    if (strlen(m[ns])=  ) break;
  }
  sortMatrix(m,ns);
  printf(”sorted strings:\n");
  for (i=0; i<ns; i++)
  printf("%s\n", m[i]);
}
```



Vector of pointers

Double level of storage

- vector `buf` of 520 elements that contain the strings (including the terminator), one after the other
- vector `v` of 20 pointers to the first element of each string
  - `v[0]` is equivalent to `buf` (or `&buf[0]`), the other pointers are computed based on the length of the string

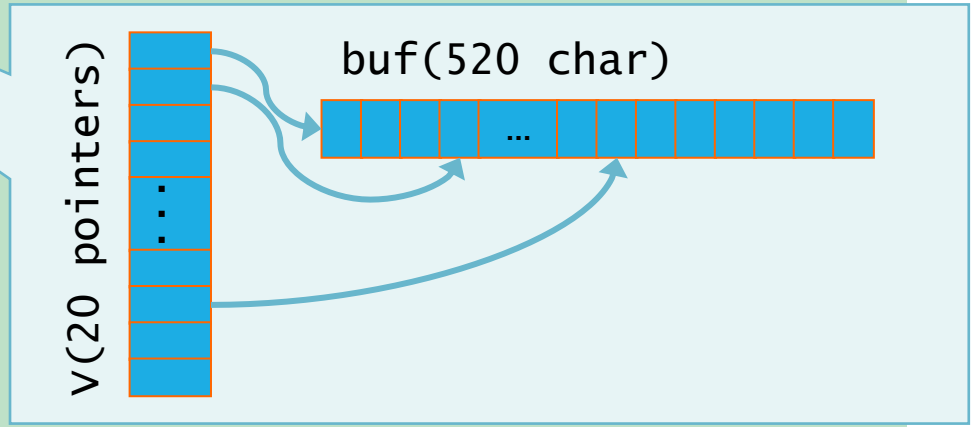


Solution 2: vector of pointers

```
void main (void) {
     int i,ns;
     char *v[20], buf[520];
     printf(”Insert the strings:\n");
     for (ns=i=0; ns<20; ns++) {
            v[ns]=buf+i; gets(v[ns]);
            if (strlen(v[ns])==0) break;
             i = i+strlen(v[ns])+1;
     }
     sortVector(v,ns);
     printf(”sorted strings:\n");
     for (i=0; i<ns; i++)
       printf("%s\n", v[i]);
}
```

Solution 2: vector of pointers

```
void main (void) {
     int i,ns;
     char *v[20], buf[520];
     printf("Insert the strings:\n");
     for (ns=i=0; ns<20; n     ) {
            v[ns]=buf+i; gets(v
            if (strlen(v[ns])==0) br
            i = i+strlen(v[ns])+1;
     }
     sortVector(v,ns);
     printf("sorted strings:\n");
     for (i=0; i<ns; i++)
       printf("%s\n", v[i]);
}
```



Solution 2: vector of pointers

```
void main (void) {
     int i,ns;
     char *v[20], buf[520];
     printf("Insert the strings:\n");
     for (ns=i=0; ns<20; ns++) {
           v[ns]=buf+i; gets(v[ns]);
           if (strlen(v[ns])==0) break;
           i = i+strlen(v[ns])+1;
     }
     sortVector(v,ns);
     printf("sorted strings:\n");
     for (i=0; i<ns; i++)
       printf("%s\n", v[i]);
}
```

buf+i = &buf[i]

Solution 2: vector of pointers

```
void main (void) {
     int i,ns;
     char *v[20], buf[520];
     printf("Insert the strings:\n");
     for (ns=i=0; ns<20; ns++) {
      v[ns]=buf+i; gets(v[ns]);
      if (strlen(v[ns])==0) break;
      i = i+strlen(v[ns])+1;
     }
     sortVector(v,ns);
     printf("sorted strings:\n");
     for (i=0; i<ns; i++)
       printf("%s\n", v[i]);
}
```

Proceed in `buf` by jumping over the current string (and terminator '\0')

Solution 2: vector of pointers

```
void main (void) {
     int i,ns;
     char *v[20], buf[520];
     printf("Insert the strings:\n");
     for (ns=i=0; ns<20; ns++) {
      v[ns]=buf+i; gets(v[ns]);
      if (strlen(v[ns])==0) break;
      i = i+strlen(v[ns])+1;
     }
     sortVector(v,ns);
     printf("sorted strings:\n");
     for (i=0; i<ns; i++)
       printf("%s\n", v[i]);
}
```

v[i] points at buf[i] → this instruction prints the i-th string in buf

Comparing the two solutions

- Matrix of characters
  - 20 rows: maximum number of strings
  - 51 columns: maximum length of a string
  - 20*51 = 1020 characters: dimension of the matrix
- Vectors of pointers
  - 20 pointers: dimension of the vector of pointers
  - 520 characters: dimension of the vector of characters (500 characters for the strings + 20 terminators)

Comparing the two solutions

- Matrix of characters
  - 20 rows: maximum number of strings
  - 51 columns: maximum length of a string
  - 20*51 = 1020 characters: dimension of the matrix
- Vectors of pointers
  - 20 pointers: dimension of the vector of pointers
  - 520 characters: dimension of the vector of characters (500 characters for the strings + 20 terminators)

20 pointers + 520 characters < 1020 characters !

Sorting with matrix (selection sort)

```
void sortMatrix (char m[][51], int n) {
  int i, j, min; char tmp[51];
  for (i=0; i<n-1; i++) {
    min = i;
    for (j=i+1; j<n; j++)
      if (strcmp(m[min],m[j])>0)
        min = j;
    strcpy(tmp,m[i]);
    strcpy(m[i],m[min]);
    strcpy(m[min],tmp);
  }
}
```

Sorting with matrix (selection sort)

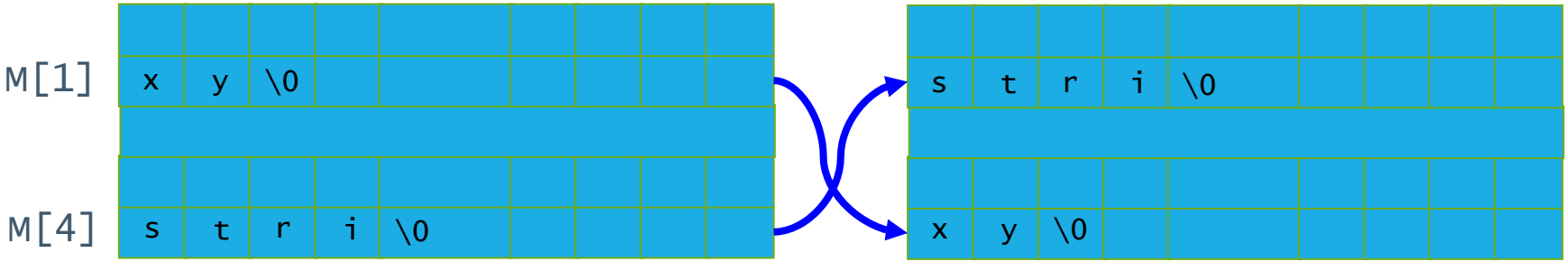
```
void sortMatrix (char m[][51], int n) {
  int i, j, min; char tmp[51];
  for (i=0; i<n-1; i++) {
    min = i;
    for (j=i+1; j<n; j++)
      if (strcmp(m[min],m[j])>0)
        min = j;
    strcpy(tmp,m[i]);
    strcpy(m[i],m[min]);
    strcpy(m[min],tmp);
  }
}
```

With a matrix of characters the swaps of rows are implemented with strcpy

Example

Swap rows 1 and 4, containing "xy" and "stri"

- Solution with matrix of characters



Sorting with array of pointers

```
void sortVector (char *m[], int n){
  int i, j, min; char *tmp;
  for (i=0; i<n-1; i++) {
    min = i;
    for (j=i+1; j<n; j++)
      if (strcmp(m[min],m[j])>0)
        min = j;
    tmp = m[i];
    m[i] = m[min];
    m[min] = tmp;
  }
}
```

Sorting with vector of pointers

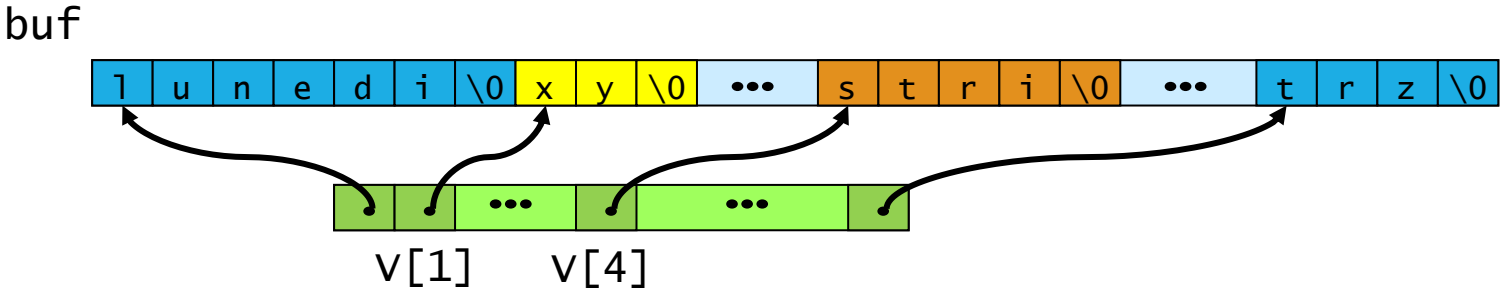
```
void sortVector (char *m[], int n){
  int i, j, min; char *tmp;
  for (i=0; i<n-1; i++) {
    min = i;
    for (j=i+1; j<n; j++)
      if (strcmp(m[min],m[j])>0)
        min = j;
    tmp = m[i];
    m[i] = m[min];
    m[min] = tmp;
  }
}
```

With vector of pointers the swaps are implemented as a swap of pointers, with = (NO strcpy)

Example

Swap rows 1 and 4, containing "xy" and "stri"

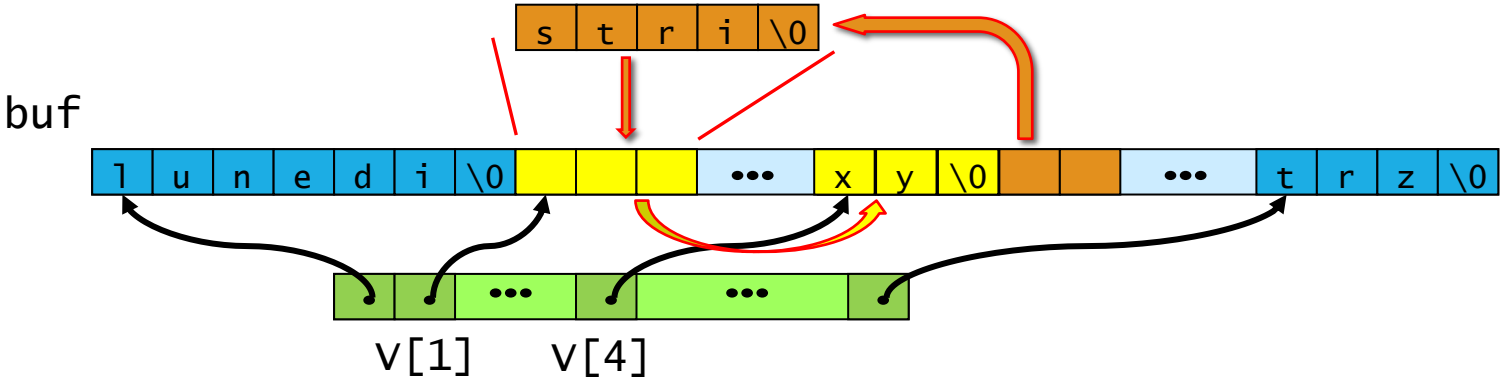
- Solution with vector of pointers



Example

Swap rows 1 and 4, containing "xy" and "stri"

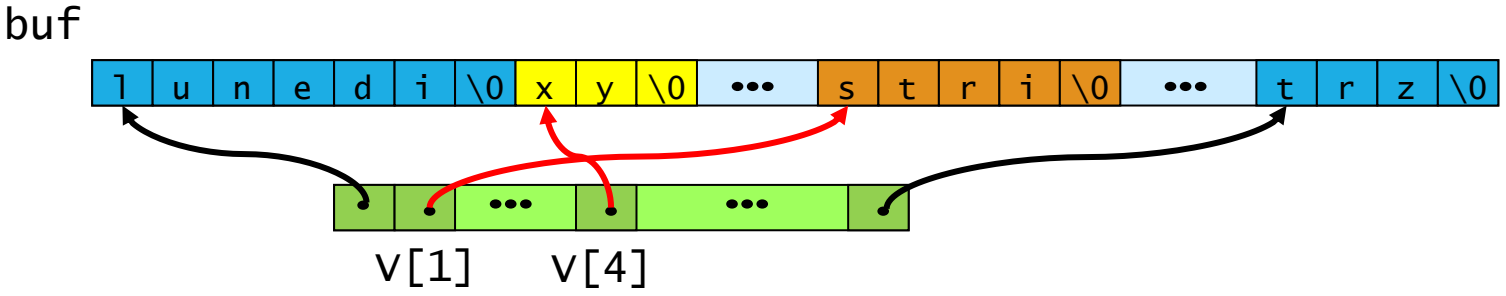
- Solution with vector of pointers
- strcpy(v[1],v[4]) cannot work!!! THERE IS NO SPACE ENOUGH!!!



Example

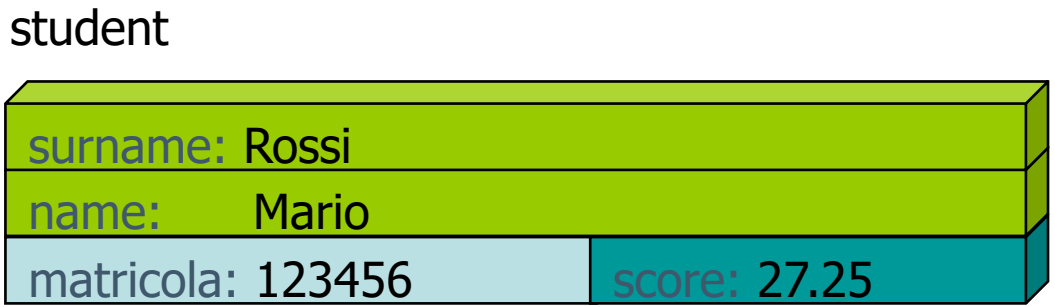
Swap rows 1 and 4, containing "xy" and "stri"

- Solution with vector of pointers
- We swap pointers:
  tmp=v[1]; v[1]=v[4]; v[4]=tmp;



`struct`, pointers and arrays

- Heterogeneous information about an object can be aggregated as fields of an individual aggregated datum



Recap of `struct`

- Aggregated data type in C is called `struct`. It is the equivalent of a `record` in other languages
- A `struct` (structure) è is composed by fields:
  - Fields are either basic data types or other structs
  - Each field in a struct can be accessed by means of its identifier (unlike arrays, where elements are accessed by indexing)

Example

```
struct student {
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
```

```
struct student
{
    char surname[MAX], name[MAX];
    int matricula;
    float score;
};
```

A new data type

- The new type is `struct student`
- Keyword `struct` is mandatory

```
struct student
{
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
```

Name of the struct

- Same rules as for the names of the variables
- Names of `struct` need to be different from the names of other struct (they can be the same as the name of other variables, but better avoid…

```
struct student
{
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
```

Fields

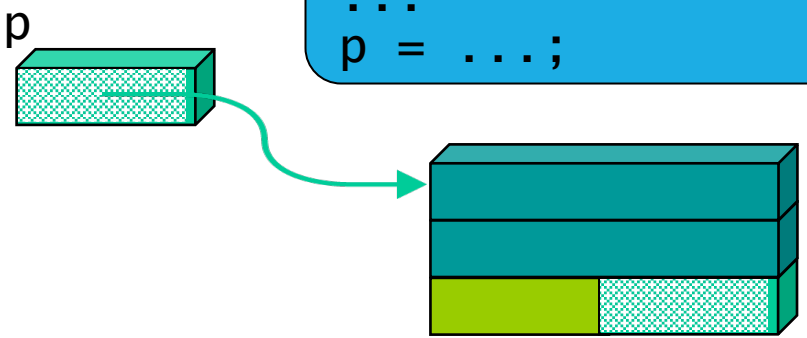
- Fields correspond to local variables of the struct
- Each field has a type and an identifier

Pointer to `struct`

- To access a `struct` using pointers, the same rules of the other data types apply
- Note that a pointer can:
  - Point to a whole struct
  - Point to a field of a struct
  - Be a field of a struct

Pointer to a whole struct

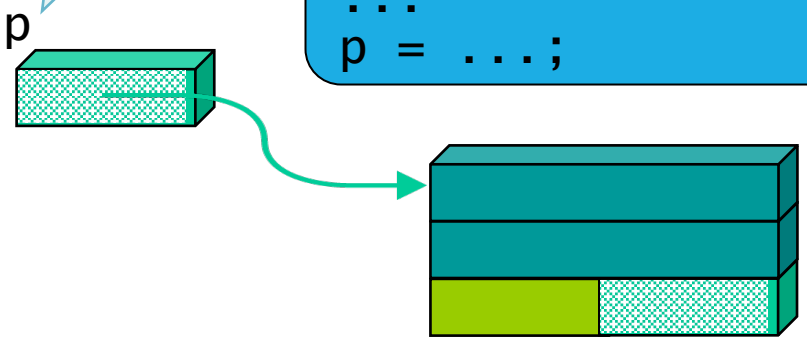
```
struct student {
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
...
struct student *p;
...
p = ...;
```



Pointer to a whole struct

```
struct student {
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
...
struct student *p;
...
p = ...;
```

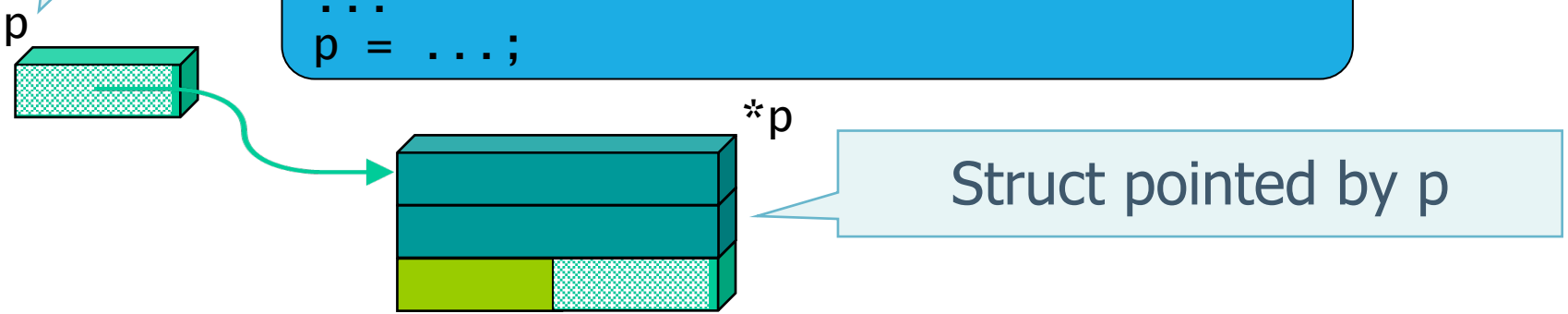
Pointer variable



Pointer to a whole struct

```
struct student {
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
...
struct student *p;
...
p = ...;
```

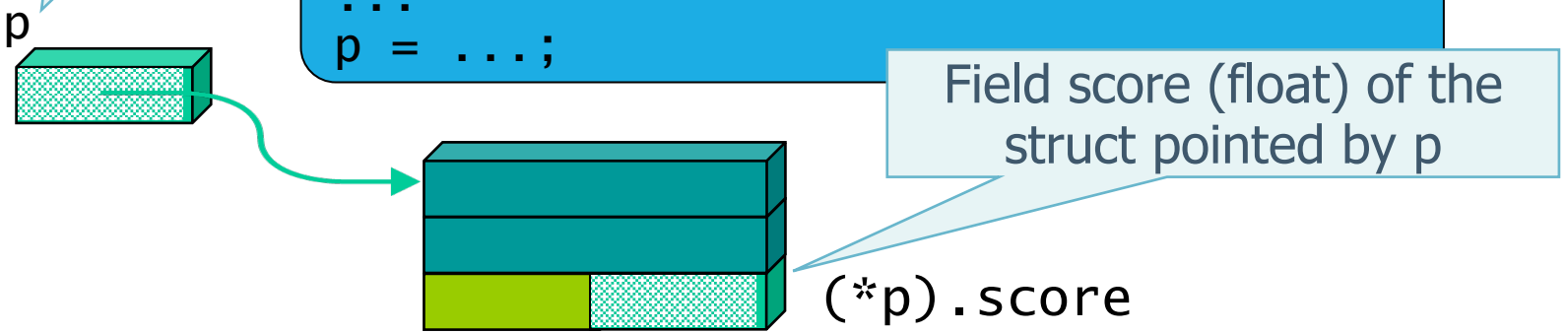
Pointer variable



Pointer to a whole struct

```
struct student {
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
...
struct student *p;
...
p = ...;
```

Pointer variable

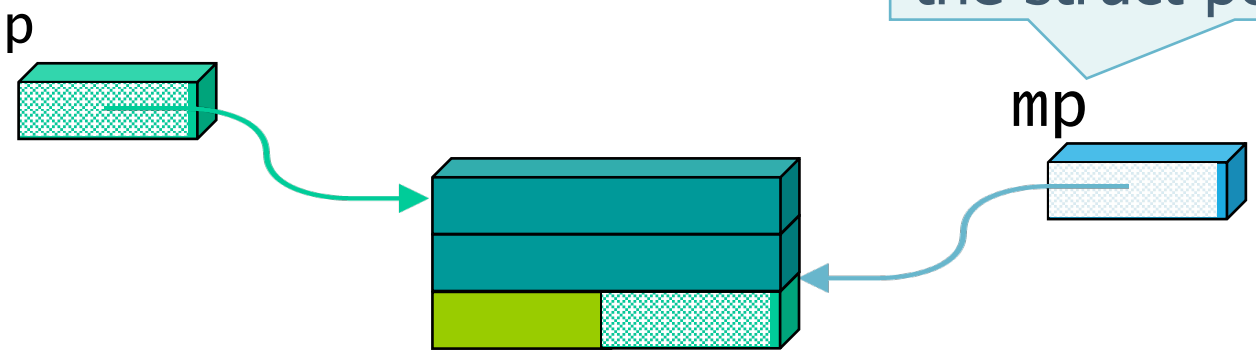


Field score (float) of the struct pointed by p

Pointer to a field of a struct

```
...
struct student *p;
float *mp;
...
p = ...;
mp = &(*p).score;
```

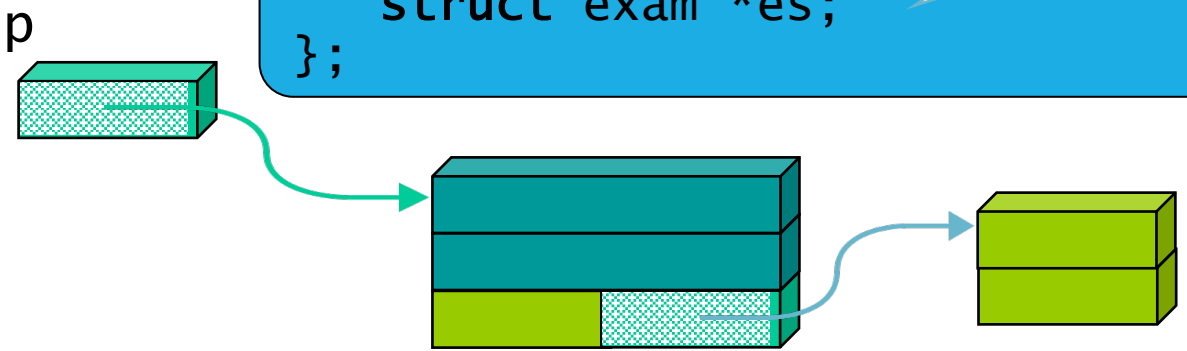
Pointer to the field score of the struct pointed by p



Pointer as a field of a struct

```
struct exam {
    int written, oral;
};
struct student {
    char surname[MAX], name[
    int matricola;
    struct exam *es;
};
```

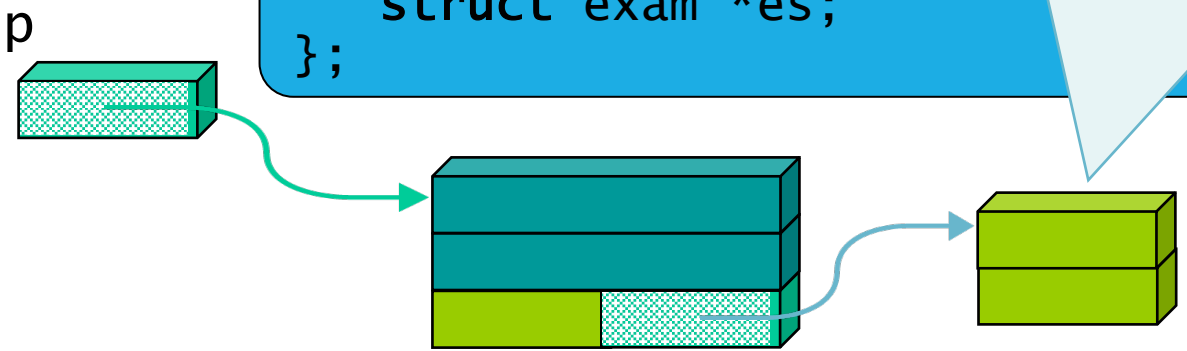
Note that it is NOT an internal struct:
`struct exam es;`
It is a pointer:
`struct esame *es;`



Pointer as a field of a struct

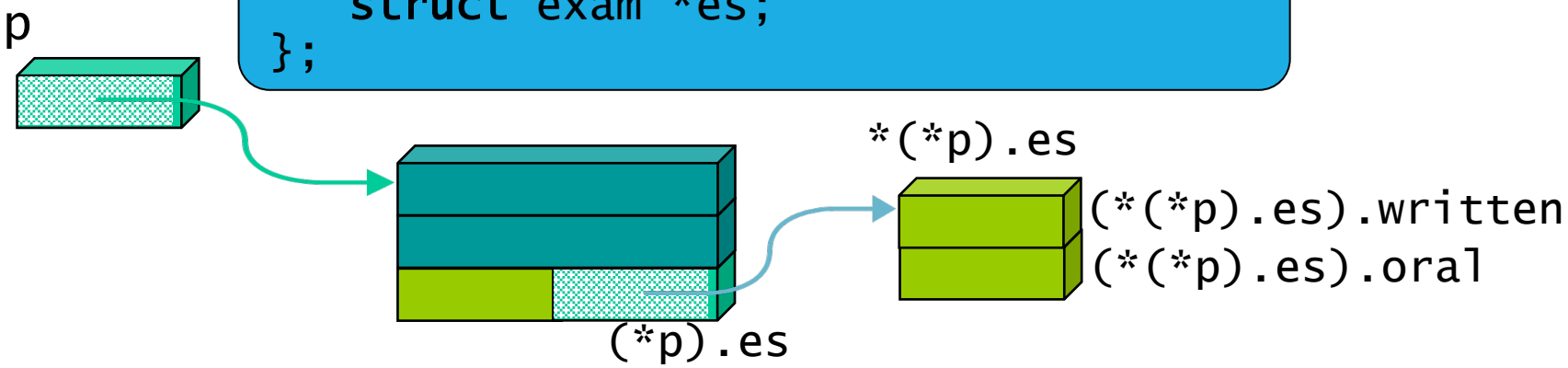
```
struct exam {
   int written, oral;
};
struct student {
   char surname[MAX],
   int matricola;
   struct exam *es;
};
```

The pointed struct exam
- Is OUTSIDE of `struct student`
- It exists independently from the other struct
- It is NOT generated AUTOMATICALLY



Pointer as a field of a struct

```
struct exam {
   int written, oral;
};
struct student {
   char surname[MAX], name[MAX];
   int matricola;
   struct exam *es;
};
```



Access to a pointed structure

- C language provides an ***alternative notation*** (more compact) to represent the fields of a pointed structure
  - Instead of:

```
(*p).score
(*(*p).exam).written
```

  - You can write:

```
p->score
p->exam->written
```

Access to a pointed structure

- C language provides an ***alternative notation*** (more compact) to represent the fields of a pointed structure
  - Instead of:

```
(*p).score
(*(*p).exam).written
```

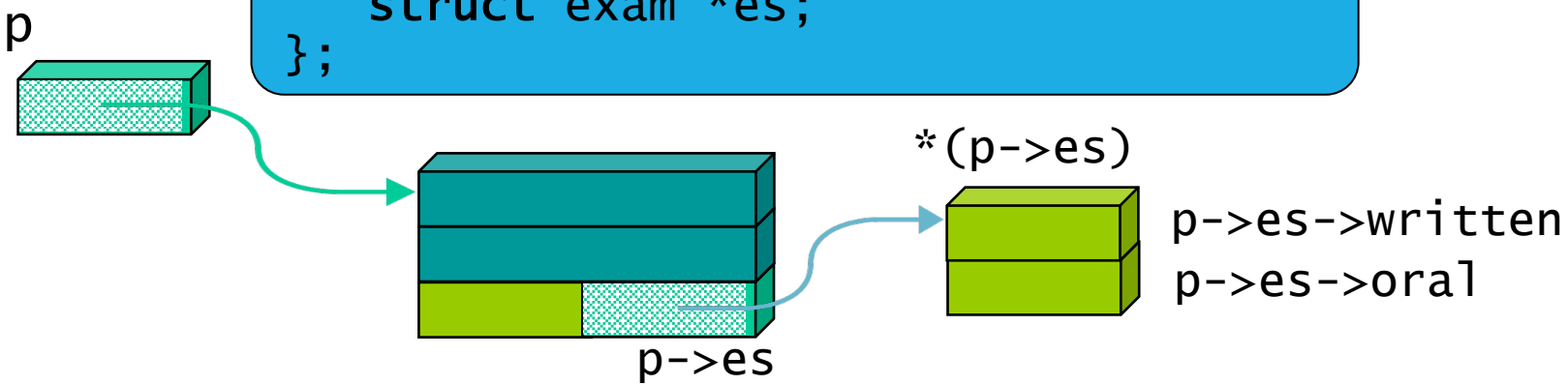
  - You **can** write:

```
p->score
p->exam->written
```

You see "you can" but you should interpret as **"you must"**:
- The notation with () and * is very difficult to read
- The vast majority of the programmers use the -> notation

Pointer as a field of a struct

```
struct exam {
   int written, oral;
};
struct student {
   char surname[MAX], name[MAX];
   int matricola;
   struct exam *es;
};
```



Recursive `Struct`

- A **recursive** struct is a `struct` that has one of more pointers to structs of the same type among its fields
- Used to generate lists, trees, graphs
- We give a first definition for the sake of completeness: we will not use them in this course!

Example: solution 1

```
struct student {
  char surname[MAX], name[MAX];
  int matricola;
  struct student *link;
};
```

Recursive `Struct`

- A **recursive** struct is a `struct` that has one of more pointers to structs of the same type among its fields
- Used to generate lists, trees, graphs
- We give a first definition for the sake of completetess: we will not use them in this course!

Example: solution 1

```
struct student {
  char surname[MAX], name[MAX];
  int matricola;
  struct student *link;
};
```

field `link` is a pointer to a `struct student`

Recursive `Struct`

- A **recursive** struct is a `struct` that has one of more pointers to structs of the same type among its fields
- Used to generate lists, trees, graphs
- We give a first definition for the sake of completetess: we will not use them in this course!

Example: solution 1

```
struct student {
  char surname[MAX], name[MA
  int matricola;
  struct student *link;
};
```

It WON'T point to itself, but to another struct of the same type

Recursive `Struct`

- A **recursive** struct is a `struct` that has one of more pointers to structs of the same type among its fields
- Used to generate lists, tree
- We give a first definition fo
  use them in this course!

Exception to the rule: you "use" `struct student` (to define a pointer) BEFORE you have finished defining struct student (only after **};**)

Example: solution 1

```
struct student {
  char surname[MAX]   name[MAX];
  int matricola;
  struct student *link;
};
```

Recursive `Struct`

- A **recursive** struct is a `struc` structs of the same type amo
- Used to generate lists, trees,
- We give a first definition for t use them in this course!

Exception to the rule: you "use" `struct student` (to define a pointer) BEFORE you have finished defining struct student (only after **};**)
It can be done only with pointers (not with other data types) because the DIMENSION of a pointer is apriori known (32 or 64 bit, based on the processor)

Example: solution 1

```
struct student {
  char surname[MAX], name[MAX];
  int matricola;
  struct student *link;
};
```

Example: solution 2

New type pointer to `struct student` (used even before starting its definition!): ok because the dimension is known

```
typedef struct student *P_stud;
struct student {
  char surname[MAX], name[MAX];
  int matricola;
  P_stud link;
};
```

field `link` of type `P_stud`

Example: solution 3

New type T_stud
(of unknown dimension)

```
typedef struct student T_stud;
struct student {
  char surname[MAX], name[MAX];
  int matricola;
  T_stud *link;
};
```

Example: solution 3

New type T_stud
(of unknown dimension)

```
typedef struct student T_stud;
struct student {
  char surname[MA
  int matricola;
  T_stud *link;
};
```

Before defininig the struct student, it won't be allowed to declare variables (or fields) of type `T_stud`, because the dimension is not known

Example: solution 3

New type T_stud
(of unknown dimension)

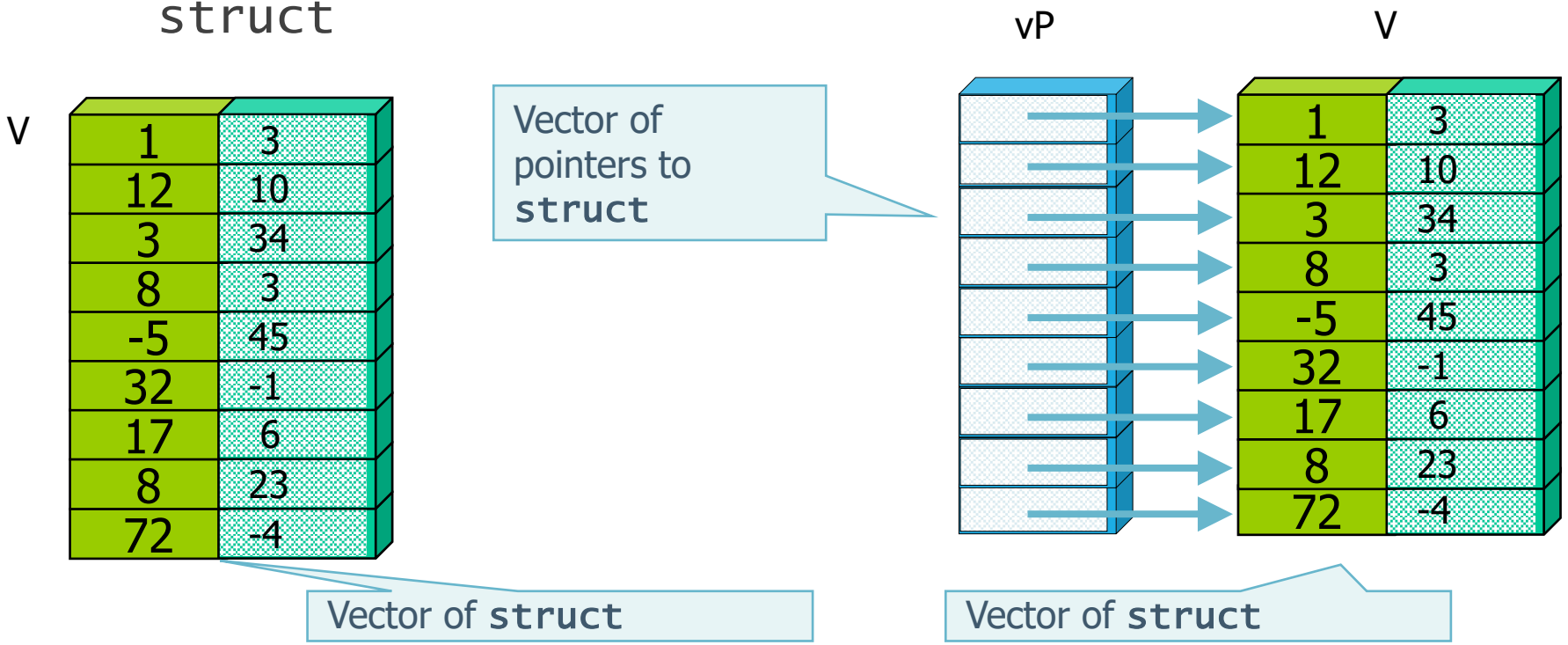
```
typedef struct student T_stud;
struct student {
  char surname[MA
  int matricola;
  T_stud *link;
};
```

Before defininining the struct student, it won't be allowed to declare variables (or fields) of type `T_stud`, because the dimension is not known

field `link` of type pointer to `T_stud` has a known dimension because it is a pointer

Vectors of pointers to `struct`

- A vector of `struct` is different from a vector of pointers to `struct`

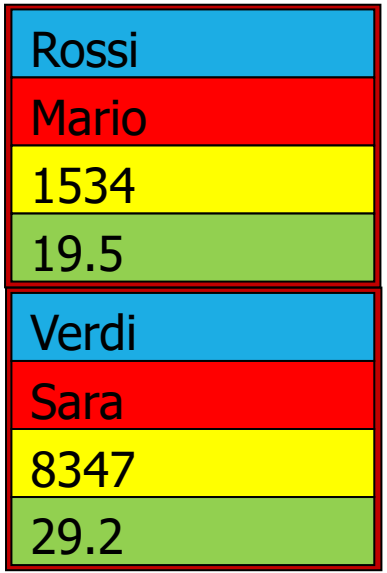


Example

- Type struct

```
typedef struct student {
  char surname[MAXS];
  char name[MAXS];
  int matr;
  float score;
} stud_t;
```

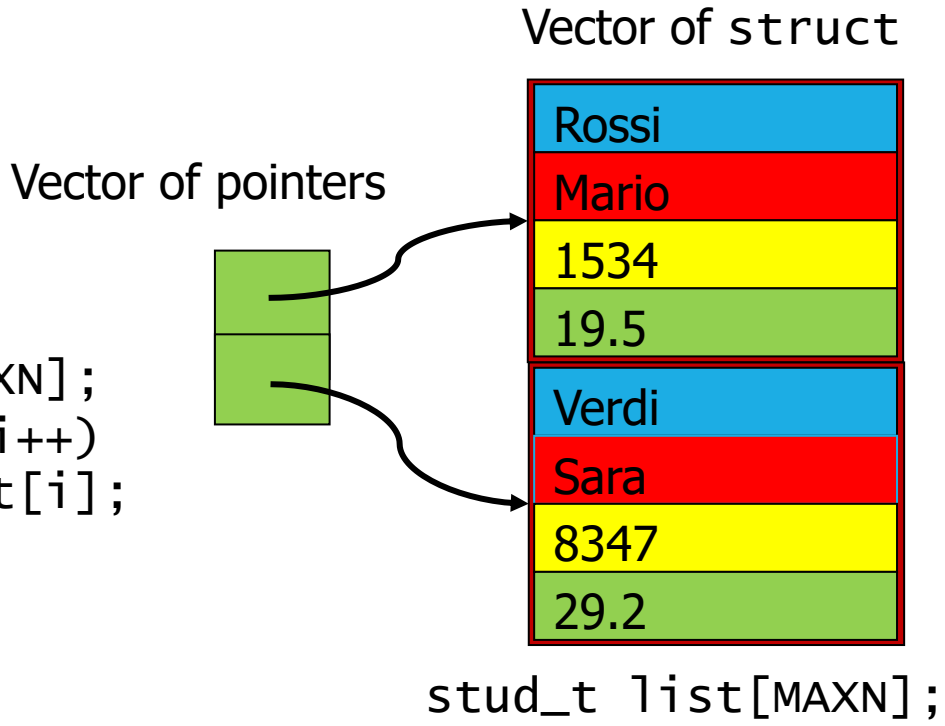
Vector of `struct`




```
stud_t list[MAXN];
```

Vector of pointers to `struct`

```
stud_t *listRef[MAXN];
for (i=0; i<MAXN; i++)
 listRef[i] = &list[i];
```



Example

- Write a program that:
  - acquires a list of students from a file whose name is received as the first argument to main
  - sorts the list by increasing matricola ids
  - Does some processing (not defined…) on the sorted list
  - writes the processed list to a second file, whose name is received as the second argument.

Solution1: vector of `struct` (swap of values)

```c
/* ... #include and #define */
typedef struct student {
  char surname[MAXS]; char name[MAXS];
  int matr; float score;
} stud_t;
/* ... prototypes */
int main(int argc, char *argv[]) {
  stud_t list[MAXN];
  int ns = readStud(argv[1],list,MAXN);
  sortStudByMatr(list,ns);
  processSortedList(list,ns);
  writeStud(argv[2],list,ns);
  return 0;
}
```

```c
int readStud(char *nameFile, stud_t *el, int nmax) {
  int n;
  FILE *fp = fopen(nameFile,"r");
  for (n=0; n<nmax; n++) {
    if (fscanf(fp,"%s%s%d%f", el[n].surname,
                  el[n].name, &el[n].matr,
                  &el[n].score)==EOF) break;
  }
  fclose(fp);
  return n;
}
```

Solution1: vector of `struct` (swap of values)

```
/* ... #include and #define */
typedef struct student {
  char surname[MAXS]; char name[MAXS];
  int matr; float score;
} stud_t;
/* ... prototypes */
int main(int argc, char *argv[]) {
  stud_t list[MAXN];
  int ns = readStud(argv[1],list,MAXN);
  sortStudByMatr(list,ns);
  processSortedList(...);
  writeStud(argv[2],list,ns);
  return 0;
}
```

```
int readStud(char *nameFile, stud_t *el, int nmax) {
  ...
  while (fscanf(fp,"%s%s%d%f", el[n].surname,
                el[n].name, &el[n].matr,
                &el[n].score)==EOF) break;
  }
  fclose(fp);
  return n;
}
```

Maximum dimension of the vector

Actual dimension of the vector (as obtained from the file)

Solution1: vector of `struct` (swap of values)

```
void writeStud(char *nameFile, stud_t *el,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp, "%s %s %d %f\n", el[i].surname,
                el[i].name, el[i].matr,
                el[i].score);
  }
  fclose(fp);
}
// comparison of struct received by value
int compareMatr(stud_t s1, stud_t s2) {
  return s1.matr-s2.matr;
}
```

```
// comparison for struct received by reference/pointer
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortStudByMatr(stud_t *el, int n) {
  stud_t temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatr(el[j],el[imin])<0)
        imin = j;
    temp = el[i]; el[i] = el[imin]; el[imin] = temp;
  }
}
```

Solution1: vector of `struct` (swap of values)

```
void writeStud(char *nameFile, stud_t *el,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp
  }
  fclose(fp);
}
// comparison of struct received by value
int compareMatr(stud_t s1, stud_t s2) {
  return s1.matr-s2.matr;
}
```

passage by value:
The function receives a "copy" of the struct to be compared

selection sort

```
// comparison for struct received by reference/pointer
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortStudByMatr(stud_t *el, int n) {
  stud_t temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatr(el[j],el[imin])<0)
        imin = j;
    temp = el[i]; el[i] = el[imin]; el[imin] = temp;
  }
}
```

Solution1: vector of `struct` (swap of values)

```
void writeStud(char *nameFile, stud_t *el,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<
    fprintf(fp
  }
  fclose(fp);
}
// comparison of struct received by value
int compareMatr(stud_t s1, stud_t s2) {
  return s1.matr-s2.matr;
}
```

ALTERNATIVE:
passage by reference/pointer:
The function receives the pointers of the structs to be compared

```
// comparison for struct received by reference/pointer
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortStudByMatr(stud_t *el, int n) {
  stud_t temp;
  int i, j, imin;
    (i=0; i<n-1; i++) {
    imi    i;
    for (j =  1; j < n; j++)
      if (compareMatrByRef(&el[j],&el[imin])<0)
        imin = j;
    temp = el[i]; el[i] = el[imin]; el[imin] = temp;
  }
}
```

selection sort

Solution1: vector of `struct` (swap of values)

```
void writeStud(char *nameFile, stud_t *el,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp, "%s %s %d %f\n", el[i].surname,
                el[i].name, el[i].matr,
                el[i].score);
  }
  fclose(fp);
}
// comparison of struct received by value
int compareMatr(stud_t s1, stud_t s2) {
  return s1.matr-s2.matr;
}
```

```
// comparison for struct received by reference/pointer
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortStudByMatr(stud_t *el, int n) {
  stud_t temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatr(el[j],el[imin])<0)
        imin = j;
    temp = el[i]; el[i] = el[imin]; el[imin] = temp;
  }
}
```

Solution 2: vector of pointers to `struct` (swap of pointers)

```
/* ... #include and #define */
/* ... Typedef struct ... */
/* ... prototypes */
int main(int argc, char *argv[]) {
  stud_t list[MAXN], *listRef[MAXN];
  int i, ns = readStud(argv[1],list,MAXN);
  for (i=0; i<ns; i++)
    listRef[i]=&list[i];
  sortRefStudByMatr(listRef,ns);
  processSortedRef(listRef,ns);
  writeRefStud(argv[2],elencoRif,ns);
  return 0;
}
/* ... readStud is the same */
```

```
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortRefStudByMatr(stud_t **elR, int n) {
  stud_t *temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByRef(elR[j],elR[imin])<0)
        imin = j;
    temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;
  }
}
```

Solution 2: vector of pointers to `struct` (swap of pointers)

```
/* ... #include and #define */
/* ... Typedef struct ... */
/* ... prototypes */
int main(int argc, char *argv[]) {
  stud_t list[MAXN], *listRef[MAXN];
  int i, ns = readStud(argv[1],list,MAXN);
  for (i=0; i<ns; i++)
    listRef[i]=&list[i];
  sortRefStudByMatr(listRef,ns);
  processSortedRef(listRef,ns);
  writeRefStud(argv[2],listRef,ns);
  return 0;
}
/* ... readStud is th
```

```
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {

{
  stud_t *temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByRef(elR[j],elR[imin])<0)
        imin = j;
    temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;
```

Maximum dimension of the vector

Actual dimension of the vector (as obtained from the file)

Solution 2: vector of pointers to `struct` (swap of pointers)

```c
/* ... #include and #define */
/* ... Typedef struct ... */
/* ... prototypes */
int main(int argc, char *argv[]) {
  stud_t list[MAXN], *listRef[MAXN];
  int i, ns = readStud(argv[1],list,MAXN);
  for (i=0; i<ns; i++)
    listRef[i]=&list[i];
  sortRefStudByMatr(listRef,ns);
  processSortedRef(listRef,ns);
  writeRefStud(argv[2],elencoRif,ns);
  return 0;
}
/* ... readStud is the same */
```

First load vector of struct

```c
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {

                                          nt n) {
  stud_t *temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByRef(elR[j],elR[imin])<0)
        imin = j;
    temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;
  }
}
```

Solution 2: vector of pointers to `struct` (swap of pointers)

```c
/* ... #include and #define */
/* ... Typedef struct ... */
/* ... prototypes */
int main(int argc, char *argv[]) {
  stud_t list[MAXN], *listRef[MAXN];
  int i, ns = readStud(argv[1],list,MAXN);
  for (i=0; i<ns; i++)
    listRef[i]=&list[i];
  sortRefStudByMatr(listRef,ns);
  processSortedRef(listRef,ns);
  writeRefStud(argv[2],elencoRif,ns);
  return 0;
}
/* ... readStud is the same */
```

```c
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortRefStudByMatr(stud_t **elR, int n) {
  stud_t *temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByRef(elR[j],elR[imin])<0)
        imin = j;
    temp = elR[i];
    elR[i] = elR[imin];
    elR[imin]=temp;
  }
}
```

Then "hooks" the pointers to the structs

Solution 2: vector of pointers to `struct` (swap of pointers)

```
/* ... #incl
/* ... Typed
/* ... proto
int main(int a
  stud_t list
  int i, ns =
  for (i=0; i<
    listRef[i]
  sortRefStudByMatr(listRef,ns);
  processSortedRef(listRef,ns);
  writeRefStud(argv[2],elencoRif,ns);
  return 0;
}
/* ... readStud is the same */
```

The sorting function receives ONLY the array of pointers to struct
`stud_t **elR`
is the same as
`stud_t *elR[]`

```
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}

void sortRefStudByMatr(stud_t **elR, int n) {
  stud_t *temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByRef(elR[j],elR[imin])<0)
        imin = j;
    temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;
  }
}
```

Solution 2: vector of pointers to `struct` (swap of pointers)

```
/* ... #include and #define */
/* ... Typedef struct ... */
/* ... prototypes */
in
    listRef[i]=&list[i];
  sortRefStudByMatr(listRef,ns);
  processSortedRef(listRef,ns);
  writeRefStud(argv[2],elencoRif,ns);
  return 0;
}
/* ... readStud is the same */
```

Compare struct starting from pointers, Swaps the pointers

```
int compareMatrByRef(stud_t *ps1, stud_t *ps2) {
  return ps1->matr-ps2->matr;
}
void sortRefStudByMatr(stud_t **elR, int n) {
  stud_t *temp;
  int i, j, imin;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByRef(elR[j],elR[imin])<0)
        imin = j;
    temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;
  }
}
```

Solution 2: vector of pointers to `struct` (swap of pointers)

```
void writeRefStud(char *nameFile, stud_t **elR,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp, "%s %s %d %f\n", elR[i]->surname,
                elR[i]->name, elR[i]->matr,
                elR[i]->score);
  }
  fclose(fp);
}
```

Solution 2: vector of pointers to `struct` (swap of pointers)

```
void writeRefStud(char *nameFile, stud_t **elR,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp, "%s %s %d %f\n", elR[i]->surname,
                elR[i]->name, elR[i]->matr,
                elR[i]->score);
  }
  fclose(fp);
}
```

Like writeStud, but with vector of pointers:
`stud_t **elR`
is the same as
`stud_t *elR[]`

Solution 2: vector of pointers to `struct` (swap of pointers)

```
void writeRefStud(char *nameFile, stud_t **elR,int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp, "%s %s %d %f\n", elR[i]->surname,
                elR[i]->name, elR[i]->matr,
                elR[i]->score);
  }
  fclose(fp);
}
```

… hence, we use -> instead of . to access to the fields of the struct

Vector of pointers to `struct:` advantages

```
/* ... Omitted portions */
  stud_t el[MAXN],
  *elRef0[MAXN], *elRef1[MAXN], *elRef2[MAXN];
  int i, ns = readStud(argv[1],el,MAXN);
  for (i=0; i<n; i++)
    elRef0[i] = elRef1[i] = elRef2[i] = &el[i];
  sortRefStudByMatr(elRef0,ns);
  sortRefStudBySurn(elRef1,ns);
  sortRefStudScore(elRef2,ns);
...
...
...
```

Vector of pointers to `struct`: advantages

```
/* ... Omitted portions */
  stud_t el[MAXN],
  *elRef0[MAXN], *elRef1[MAXN], *elRef2[MAXN];
  int i, ns = readStud(argv[1],el,MAXN
  for (i=0; i<n; i++)
    elRef0[i] = elRef1[i] = elRef2[i]
  sortRefStudByMatr(elRef0,ns);
  sortRefStudBySurn(elRef1,ns);
  sortRefStudScore(elRef2,ns);
...
...
...
```

Different sortings possible at the same time

Vector of pointers to `struct:` advantages

```
/* ... Omitted portions */
  stud_t el[MAXN];
  *elRef0[MAXN], *elRef1[MAXN], *elRe
  int i, ns = readStud(argv[1],el,MAXN);
  for (i=0; i<n; i++)
    elRef0[i] = elRef1[i] = elRef2[i] = &el[i];
  sortRefStudByMatr(elRef0,ns);
  sortRefStudBySurn(elRef1,ns);
  sortRefStudScore(elRef2,ns);
...
...
...
```

Only one vector of struct:
data are NOT REPLICATED

Vector of pointers to `struct:` advantages

```
/* ... Omitted portions */
  stud_t el[MAXN],
 *elRef0[MAXN], *elRef1[MAXN], *elRef2[MAXN];
  int i, ns = readStud(argv[1],el,MAXN);
  for (i=0; i<n; i++)
    elRef0[i] = elRef1[i] = elRef2[i] = &
  sortRefStudByMatr(elRef0,ns);
  sortRefStudBySurn(elRef1,ns);
  sortRefStudScore(elRef2,ns);
...
...
...
```

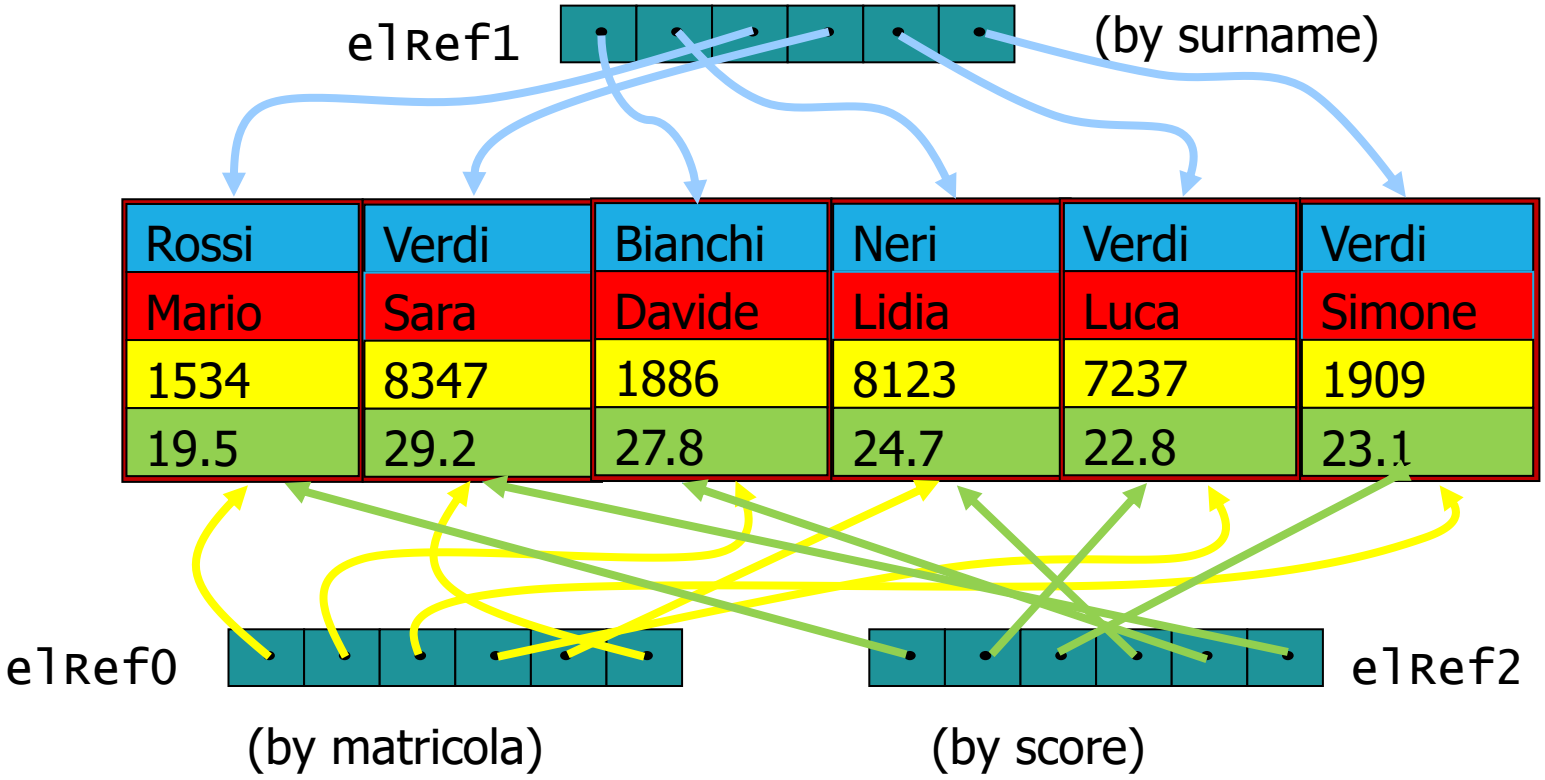
Three vectors of pointers: only pointers are REPLICATED

Vector of pointers to `struct`: advantages

```
/* ... Omitted portions */
  stud_t el[MAXN],
  *elRef0[MAXN], *elRef1[MAXN], *elRef2[MAXN];
  int i, ns = readStud(argv[1],el,MAXN);
  for (i=0; i<n; i++)
    elRef0[i] = elRef1[i] = elRef2[i] = &
  sortRefStudByMatr(elRef0,ns);
  sortRefStudBySurn(elRef1,ns);
  sortRefStudScore(elRef2,ns);
...
...
...
```

Three different sorting functions, with different sorting criteria, applied to the three vectors of pointers

In the practice



Example

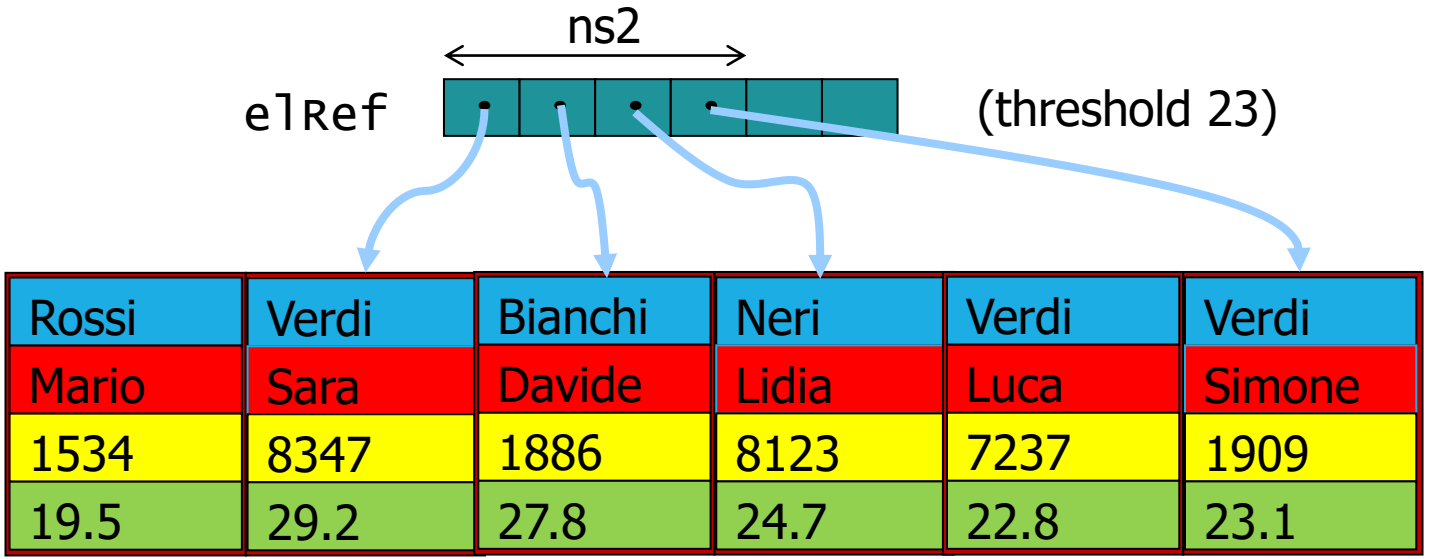
- Write a program that:
  - acquires a list of students from a file whose name is received as the first argument to main
  - filters the list based on a threshold on the scores (third argument to main)
  - writes the filtered list to a second file, whose name is received as the second argument.
- Solution: given the vector of ALL data, the vector of pointers to the selected data is generated as a result
  - ATTENTION: if we needed only to print the filtered data, A VECTOR OF POINTERS WOULD NOT BE NECESSARY. It is just a partial example!
  - The vector (result) can instead be useful for any subsequent processing (NOT PROVIDED IN THIS EXAMPLE)

```
int filterByScore(stud_t *el, stud_t **elR,
                  int n, float s);

int main(int argc, char *argv[]) {
  stud_t el[MAXN], *elRef[MAXN];
  int i, ns, ns2;
  float  threshold = atof(argv[3];
  ns = readStud(argv[1],el,MAXN);
  ns2 = filterByScore(el,elRef,ns,threshold)
  writeRefStud(argv[2],elRef,ns2);
  return 0;
}
```

```
int filterByScore(stud_t *el, stud_t **elR,
                  int n, float s){
  int i, n2;
  for (i=n2=0; i<n; i++)
    if (el[i].score>=s)
      elR[n2++] = &el[i];
  return n2;
}
```

In the practice



Pointers and indexes

- It is possible to access data in a vector by means of **indexes**
- The index "points" at a certain element → We can say that the index plays for a vector the same role that a pointer plays for memory
- **We can try to emulate the behavour of pointers by using indexes in a vector.**

Example

- Sort by matricola a vector of references (indexes) to struct student
- Vector `elInd` initialized with the current index
- Function `sortIndStudByMatr` compares data and swaps indexes if necessary
- Function `writeIndStud` access to the element to be printed by means of the index that is contained in the vector of indeces `elInd.`

```c
int main(int argc, char *argv[]) {
  stud_t el[MAXN];
  int elInd[MAXN];
  int i, ns;
  ns = readStud(argv[1],el,MAXN);
  for (i=0; i<ns; i++)
    elInd[i] = i;
  sortIndStudByMatr(el,elInd,ns);
  writeIndStud(argv[2],el,elInd,ns);
  return 0;
}
int compareMatrByInd(stud_t *el, int id1, int id2) {
  return el[id1].matr - el[id2].matr;
}
```

```c
void sortIndStudByMatr(stud_t *el, int *elI,
                       int n) {
  int i, j, imin, temp;
  for (i=0; i<n-1; i++) {
    imin = i;
    for (j = i+1; j < n; j++)
      if (compareMatrByInd(el,elI[j],
                           elI[imin])<0)
        imin = j;
    temp = elI[i];
    elI[i] = elI[imin];
    elI[imin] = temp;
  }
}
```

```
void writeIndStud(char *nameFile, stud_t *el,
                  int *elI, int n) {
  int i;
  FILE *fp = fopen(nameFile,"w");
  for (i=0; i<n; i++) {
    fprintf(fp, "%s %s %d %f\n",
         el[elI[i]].surname, el[elI[i]].name,
         el[elI[i]].matr, el[elI[i]].score);
  }
   fclose(fp);
}
```

Example: Sort by different keys

- Multiple vectors of indexes
- Compare data and swap indexes if necessary
- Access to the element to be printed by means of the index contained in the corresponding vector of indexes

```
int main(int argc, char *argv[]) {
  stud_t el[MAXN]; int i, ns;
  int elInd0[MAXN], elInd1[MAXN], elInd2[MAXN];
  ns = readStud(argv[1],el,MAXN);
  for (i=0; i<n; i++)
    elInd0[i] = elInd1[i] = elInd2[i] = i;
  sortIndStudByMatr(el,elInd0,ns);
  sortIndStudBySurname(el,elInd1,ns);
  sortIndStudByScore(el,elInd2,ns);
  // … … …
  writeIndStud(argv[2],el,elInd0,ns);
  writeIndStud(argv[3],el,elInd1,ns);
  writeIndStud(argv[4],el,elInd2,ns);
  return 0;
}
```

In the practice

`elInd1`

2	3	0	4	1	5

(by surname)

Rossi	Verdi	Bianchi	Neri	Verdi	Verdi
Mario	Sara	Davide	Lidia	Luca	Simone
1534	8347	1886	8123	7237	1909
19.5	29.2	27.8	24.7	22.8	23.1

`elInd0`

0	2	5	4	3	1

(by matricola)

0	4	5	3	2	1

`elInd2`

(by score)

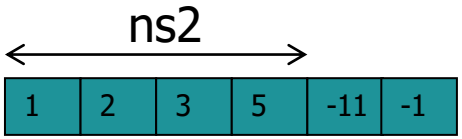
Example: filter by score

```
int main(int argc, char *argv[]) {
  stud_t el[MAXN];
  int i, ns, ns2, elInd[MAXN];
  float threshold = atof(argv[3]);
  ns = readStud(argv[1],el,MAXN);
  ns2 = filterScoreInd(el,elInd,ns,threshold);
  writeIndStud(argv[2],el,elInd,ns2);
  return 0;
}
```

```
int filterScoreInd(stud_t *el, int *elI,
                   int n, float s) {
  int i, n2;
  for (i=n2=0; i<n; i++) {
    if (el[i].score>=s)
      elI[n2++] = i;
  }
  return n2;
}
```

In practice …

`elInd`



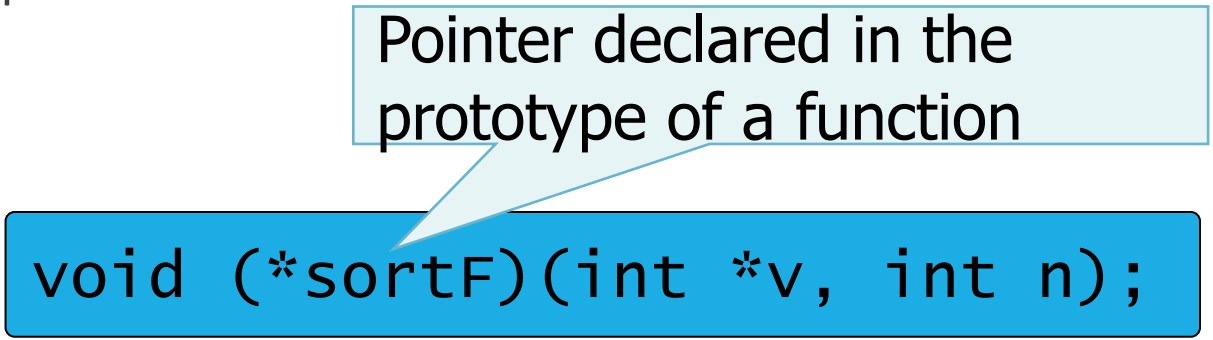
(threshold 23)

Rossi	Verdi	Bianchi	Neri	Verdi	Verdi
Mario	Sara	Davide	Lidia	Luca	Simone
1534	8347	1886	8123	7237	1909
19.5	29.2	27.8	24.7	22.8	23.1

Advanced used of pointers

Pointer to function

- We can use a pointer to point even at a function
- By doing so, we can associate the pointer to the function to be called at run-time based on a variable or on a formal parameter
- Example of declaration:



Pointer to function

- In the program, given a function:

```
void selectionSort(int *vet, int n);
```

- It is possible the assignment:

```
sortF = selectionSort;
```

- And then, the call:

```
sortF(dati, ndati);
```

The generic pointer `void *`

- It is an **opaque** pointer: a simple memory address that is not associated to any specific data type
- In assignments it is compatible with any other pointer type (explicit cast is not necessary)
- It is needed to:
  - Handle pointers to data type of which a function is not aware of
  - Reference to a datum that can switch to different types at run-time.