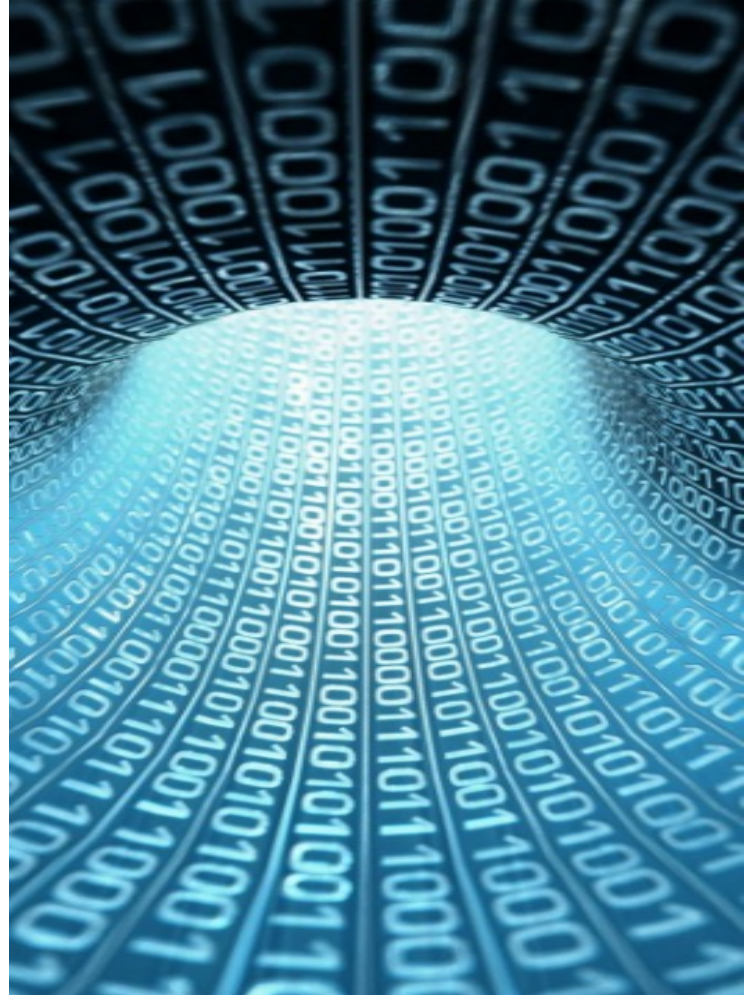




Aggregated data types

DECLARATION AND USE OF ARRAYS,
MATRICES AND STRUCT IN C



Arrays (vectors and matrices)

AGGREGATES OF HOMOGENEOUS ELEMENTS,
ACCESSIBLE BY INDEXING

Vectors (mono-dimensional arrays)

- Aggregates elements of the same type
- Definition (declaration): `<type> <identifier>[<dimension>];`
 - **Examples:** `int v[10]; char s[L]; float w[N];`
 - The dimension needs to be CONSTANT (either an integer value or an integer constant, defined with `#define` or `const int`)
 - *variable length arrays* are possible in C, but difficult to handle: not suggested
 - Only at the time of the declaration, it is possible to make an explicit initialization:
`int v[10] = {-2,0,1,10,-7,12,34,9,-3,6};`
`char s1[6] = {'h','e','l','l','o','\0'};`
`char s2[6] = "hello"; // Same effect as the previous instruction`
 - It is possible to have partial initialization or declare implicit dimensions (rules omitted here...)

Vectors (mono-dimensional arrays): usage

- Elements are identified by 0 to N-1 positions (where N is the vector's size):
 - Example: `v[0]`, `v[1]`, ..., `v[N-1]`
- There are NO atomic operations on the whole array
 - Only exception is the optional initialization during the declaration (see previous slide)
 - Atomic operations on strings can be done only with library functions (`<string.h>`) or IO functions
- There are NO atomic operations on portions of an array (no built-in slicing...)
- Operations need to be done one element at a time:
 - `v[0] = x;`
 - `s[4] = 'a';`
 - `w[i] = w[j];`
 - `s[N-j-1] = s[0];`

Vectors and functions

- When a vector is passed as parameter to a function
 - Formal parameter
 - Name of the vector with []
`int function(int vett[]);`
 - Actual parameter
 - Name of the vector, WITHOUT []
`a = function (vett);`
- The name of the vector indicates the address of the first element (& is not needed)
 - de-facto, the vector is passed “by reference”
 - The caller and the function share **the same** vector → if you modify the content of the vector in the function, the same modification is seen by the caller

Example

```
int age[20], height[20], i;
float avgAge = 0.0;
float avgHeight = 0.0;

for(i=0; i<20; i++) {
    scanf("%d %d", &age[i], &height[i]);
    avgAge += age[i];
    avgHeight += height[i];
}

avgAge = avgAge/20;
avgHeight = avgHeight /20;

... ..
```

Matrices (multidimensional arrays)

- Arrays with more than one dimension: you just need to add extra sets of []
- Definition (declaration): `<type> <identifier>[<dim1>][<dim2>]`

- **Examples:** `int M[10][10]; char s[R][C]; float W[N1][N2];`

- Dimensions need to be constant integers (same rules as vectors)

- Only at the time of the declaration, it is possible to make an explicit initialization:

```
int v[5][2] = {{-2,0,1,10,-7},{12,34,9,-3,6}};  
char days[7][10] = {"monday","tuesday","wednesday","thursday",  
                   "friday","saturday","sunday"};
```

Matrices: usage

- Elements are accessed by indexing each dimension :

`M[0][j], s[r][c], ..., w[N-1][0]`

- There are NO atomic operations on the whole matrix except initialization during declaration, same as vectors
- There are NO atomic operations on portions. Nonetheless, it is possible to identify a portion of the matrix (for example, a whole row) by omitting the last index

`char s[10][10];` → `s[r]` is the `r`-th row of the matrix → an array of `char`!

→ it is possible to do: `fgets(s[r],MAX,fin);` `scanf("%s",s[r]);` etc...

- We need to access one element at a time (same as with vectors):

`v[0][0] = x; s[4][1] = 'a'; w[i][j] = w[j][i];`

- Matrices are passed to functions "by reference", same as vectors
 - You can omit ONLY the first dimension in the formal parameter

Example

```
int matrix_diagonal[3][3] = { { 1, 0, 0 },  
                               { 0, 1, 0 },  
                               { 0, 0, 1 } } ;  
  
float M2 [N][M], V[N], Y[M];  
  
/* assignment of values to M2 and V */  
...  
/* matrix-vector multiplication*/  
for (r=0; r<N; r++) {  
    Y[r] = 0.0;  
    for (c=0; c<M; c++)  
        Y[r] = Y[r] + M2[r][c]*V[c];  
}
```

Strings

A SPECIAL FORM OF ARRAY

Strings

- They are not a specific data type, just a special vector of `char` elements
 - Example: `char nome[N];`
- They are characterized by a string terminator '`\0`' (ASCII code 0), placed after the last significant character:
 - Usually a vector is over-sized in the declaration, '`\0`' indicates where the string actually ends
- To perform operations on vectors of chars as a whole, it is necessary to use strings (with '`\0`') as operands:
 - String constants (for example: "hello") are strings: they have a '`\0`' terminator
 - IO functions: `gets/puts`, `fgets/fputs`, formatted IO with `%s`
 - Functions of the library `<string.h>`: for example, `strlen`, `strcpy`, `strcmp`, `strncmp`, `strcat`

Functions on strings `#include <string.h>`

Function	What It Does
<code>strcmp()</code>	Compares two strings in a case-sensitive way. If the strings match, the function returns 0.
<code>strncmp()</code>	Compares the first n characters of two strings, returning 0 if the given number of characters match.

<code>strcpy()</code>	Copies (duplicates) one string to another.
<code>strncpy()</code>	Copies a specific number of characters from one string to another.
<code>strlen()</code>	Returns the length of a string, not counting the or NULL character at the end of the string.

<code>strcat()</code>	Appends one string to another, creating a single string out of two.
<code>strncat()</code>	Appends a given number of characters from one string to the end of another.

Examples

```
char words[NP][MAXL], first[MAXL], last[MAXL], firstAndLast[2*MAXL];
// read words from file
int n;
for (n=0; fscanf(fin, "%s", words[n]) != EOF; n++);
// verify order
int i, sorted = 1;
for (i=1, i<n && sorted; i++) {
    if (strcmp(words[i-1], words[i]) > 0) { // if words[i-1] follows alphabetically words[i]
        sorted = 0;
    }
}
// copy first and last word
strcpy(first, words[0]); // WARNING: first = words[0] is WRONG!
strcpy(last, words[n-1]); // WARNING: last = words[n-1] is WRONG!
// join first and last
strcpy(firstAndLast, first); // copy first
strcat(firstAndLast, last); // append last
```

Strings as vectors of chars

- It is also possible to handle strings as vectors of characters, and perform operations character-by-character
- Examples:

- Remove '\n' read by fgets

```
fgets(s,MAXL,fin);
if (s[strlen(s)-1]=='\n')
    s[strlen(s)-1] = '\0';
```

- Convert a word to uppercase (toupper and tolower functions work only for single characters!)

```
void stringToUpper(char s[]) {
    int i, len = strlen(s);
    for (i=0; i<len; i++)
        s[i] = toupper(s[i]);
}
```

struct

AGGREGATES OF HETEROGENEOUS FIELDS, ACCESSIBLE
BY THEIR NAME

Struct types

- Heterogeneous aggregated type in C is a struct. Same as records of other programming languages
- A struct 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

Declaration schemes

1. Basic scheme

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

Declaration schemes

2. Contextual declaration of the new struct type and of the variables

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

Declaration schemes

3. (very rare): Contextual declaration of the type struct (without identifier) and variables

- The type `struct` is used just to declare the variables that are defined in the same context
- It will NOT be possible to declare more variables of the same struct type anywhere else in the program

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

Typedef: define a new type

- It is possible to associate an identifier to an already existing type:
 - typedef <existing type> <new name for the existing type>;
 - Example:
typedef int number;
...
number n, m;
- In practice:
 - We can use a name of our choice to refer to a type (it is especially useful to the struct types)
 - Similar to #define of literal constants (...but typedef is NOT a directive to the pre-processor, it is handled by the compiler!)

Declaration schemes

4. Synonym of `struct student`, with `typedef`

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

...
Student s, t;
```

Declaration schemes

5. Synonym of `struct student`, with `typedef` →
version without the identifier of `struct`

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

...
Student s, t;
```

Accessing fields of a struct

- After declaring a variable of struct type, the individual fields of the variable can be accessed using the operator `'.'`

<identifier of variable of type struct> . <field name>

- Example:

```
typedef struct{
    double re;
    double im;
} complex;

...

complex num1, num2;
num1.re = 0.33; num1.im = -0.43943;
num2.re = -0.133; num2.im = -0.49;
```

struct vs arrays

- Analogy:
 - They are both aggregated data types
- Differences:
 - Heterogeneous (`struct`) / Homogeneous (arrays)
 - Access by name of the field (`struct`) / by index (arrays)
 - Parameter passing by value (`struct`) / by reference (arrays)
 - Parameterized access NOT allowed (`struct`) / allowed (arrays)

Parameterized access to arrays

- Arrays are frequently used to access numbered data in a parameterized way, especially in iterative constructs
- By "parameterized" we mean "the access to the each element depends on a parameter" (for example, a variable *i*)

Example

```
for (i=0; i<N; i++) {  
    ...  
    vett[i] = ...;  
    ...  
}
```

Parameterized access to a **struct**: NO

- Fields of a **struct** CANNOT be accessed in a parameterized way



```
char field[20];  
...  
scanf("%s", field);  
printf("%s", s.field);
```

Parameterized access to a **struct**: NO

- Fields of a **struct** CANNOT be accessed in a parameterized way



```
char field[20];  
...  
scanf("%s", field);  
printf("%s", s.field);
```

field is a variable!

The fields of the struct **s** are: surname, name, matricola, score

Parameterized access to a **struct**: NO

- Possible solution: implement a function that takes the name of the field of the structure as argument



```
char field[20];  
...  
scanf("%s", field);  
printField(s, field);
```


The function HIDES the details...

```
/* in the function the access is explicit, not parameterized! */

void printField( struct student s, char id[]) {
    if (strcmp(id,"surname")==0)
        printf("%s",s.surname);
    else if(strcmp(id,"name")==0)
        printf("%s",s.name);
    else if(strcmp(id,"matricola")==0)
        printf("%d",s.matricola);
    ...
}
```

Better a struct or an array?

For homogeneous types (for example, a point in Euclidean space → aggregate of two coordinates *x* and *y*, of the same type): better ***p.x***, ***p.y*** or ***p[0]***, ***p[1]***?

Better **struct** or array?

struct suggested when:

- Not too many fields
- Fields are better identifiable by name
- We do not need parameterized access
- We would like to treat the **struct** as a whole datum (for ex. in variable assignments, or as a unique argument or a return value of a function)

Better ***p.x***, ***p.y*** !!!

Arrays can be fields of a `struct`

- A `struct` can have arrays as fields
 - Ex: surname and name in `struct student`
- Careful!
 - a `struct` is passed to functions by value, an array is passed by reference
 - If a `struct` has an array of N elements as field, passing the `struct` requires to "copy" the whole array
 - *TRICK: if you want to pass an array by value instead of by reference, you can include it as a field of a struct...*

Arrays of `struct`

We can have arrays whose elements are type `struct`

- Suggested when we have homogeneous collections of elements, where the individual elements are heterogeneous aggregates
 - Example: to handle a list of students

`struct student vet[N];` → `vett[i]` is a variable of type `struct student`!

- Careful!
 - All arrays (even the ones of type `struct`) are passed by reference to functions
 - A function can modify the content of an array of `struct`

Example

```
int main(void) {  
    struct student list[NMAX];  
    int i, n;  
  
    printf("how many students(max %d)? ", NMAX);  
    scanf("%d", &n);  
    for (i=0; i<n; i++) {  
        list[i] = readStudent();  
    }  
    sortStudents(list, n);  
    printf("students sorted by their score\n");  
    for (i=0; i<n; i++) {  
        printStudent(list[i]);  
    }  
}
```

Example

```
int main(void) {  
    struct student list[NMAX];  
    int i, n;  
  
    printf("how many students(max %d)? ", NMAX);  
    scanf("%d", &n);  
    for (i=0; i<n; i++) {  
        list[i] = readStudent();  
    }  
    sortStudents(list, n);  
    printf("students sorted by their score\n");  
    for (i=0; i<n; i++) {  
        printStudent(list[i]);  
    }  
}
```



Array of struct

Example

```
int main(void) {  
    struct student list[NMAX];  
    int i, n;  
  
    printf("how many students(max %d)? ", NMAX);  
    scanf("%d", &n);  
    for (i=0; i<n; i++) {  
        list[i] = readStudent();  
    }  
    sortStudents(list, n);  
    printf("students sorted by their score\n");  
    for (i=0; i<n; i++) {  
        printStudent(list[i]);  
    }  
}
```

Passing by reference

Example

```
int main(void) {
    struct student list[NMAX];
    int i, n;

    printf("how many students(max %d)? ", NMAX);
    scanf("%d", &n);
    for (i=0; i<n; i++) {
        list[i] = readStudent();
    }
    sortStudents(list, n);
    printf("students sorted by their score\n");
    for (i=0; i<n; i++) {
        printStudent(list[i]);
    }
}
```

```
void sortStudents(struct student el[],
                  int n) {

    /*
       this function will MODIFY the content
       of the array, by sorting the
       students based on their score.

       We will see how to implement this at
       the end of the chapter(selectionSort)
    */

}
```