# Enumerations and structures

# Enumerations

- The set type is created by defining the domain of values it contains, i.e., the list of constant values that variables of this type can take.
- Variables of this type take a value among a set. For example, for a traffic light, the color is: green, orange, or red.
- In Algorithmics, we declare a set as a type as indicated below.

- EX:

```
type color= set (blue, green, red);
var c1 : color;
C1 ← green;
```

# Enumerations

- Enums are especially useful for enhancing code readability making the code more understandable and maintainable.
- The sets can be used to iterate in loops like

```
type Color= set (Blue, green, red, white);
        var c1:color;
            for c1← blue to white do
              write (c1);
```

# Enumerations in C

- In the C language, we declare an enumeration using the 'enum' keyword.
- Syntax:
- enum new_type {list of symbols/choices/values};
- Example:
- enum color {white, blue, yellow, green, black};
- We declare a variable of type 'color' by specifying the name of the enum followed by the name of the variable to be declared (for example: enum color c1;).
- Additionally, we can assign numerical constants to each color as follows: enum color {white=10, blue=11, yellow=12, green=13, black=14};
- If numerical constants are not specified, these values start from 0 and increment by 1 for each subsequent enumerator.

# Structures

- Unlike arrays, which are data structures where all elements are of the same type, records/structures are data structures where the elements can be of different types and relate to the same semantic entity.
- The elements that compose a record are called fields or attributes.
- A record is a user-defined data type that allows grouping a finite number of elements of different types.
- A record is a complex variable that allows designating, under a single name, a set of values that can be of different types (simple or complex).

# Structures

- Before declaring a record variable, it's necessary to have previously defined the name and type of the fields that compose it.
- It's possible to create custom types and then declare variables or arrays of elements of that type.

# Declaration of Records/Structures

- The declaration of structure types occurs within a specific section of algorithms called 'Type,' which precedes the section for variables and follows the section for constants.

| Algorithm | C language |
| --- | --- |
| type <id_strucr> = structure<br><id_attribute1>:<type1>;<br><id_attribute2>:<type2>;<br>.....<br><id_attributen>:<type n>;<br>end; | typedef struct [id_structure] {<br><type1> <id_attribute1>;<br><type2> <id_attribute2>;<br><br><type n> <id_attribute n>;<br>} id_type; |

- Where <id_ch1>, <id_ch2>, ..., <id_chN> are the identifiers of the fields, and <type1>, <type2>, ..., <typeN> are their types respectively.

# Declaration of Records/Structures

- Exemple :

```
type car = structure
            brand: string;
            c: color;  /* color is an enumeration type, as shown previously */
            price: float;
        end;
 var v1 : car;
```

# Accessing the fields of a structure

- The fields of a structure are accessed by their names using the '.' operator.
- To access a field of a structure, the variable ID of the structure type is used, followed by a dot and then the name of the field you want to access.
- Syntax: <Record_Variable>.<Field Name>
- For example, to access the 'price' field of the variable 'v1' of type 'car', we write: v1.price
- Assignment: v1.price ← 2000.00;

  Or: floatVariable ← v1.price;
- Reading: read(v1.price);

# Array of structures

- In order to manage multiple entities of a given type (e.g., cars), we use a one-dimensional array structure to store these elements.
- Another example, If there is a need to record information about 100 students and manipulate their data, we declare the 'student' structure and create a vector of size 100 containing elements of type 'student'. This is depicted in the example below:

```
type student = structure;
    first_name:string;
    last_name:string;
    …
    end;
    var student_list = array [0..n] of student;
```

# Operations on structures

- Filling (read) data for a number of car structure :

```
Algorithm fill_array_car;
const n=10;
type car = structure
                    brand: string;
                    c: string;
                    price: float;
            end;
var List_car = array[0..n] of car;
i:integer;
begin
            for i← 0 to (n -1) do
                        begin
                                    read(List_car[i]. brand);
                                    read (List_car[i].c);
                                    read (List_car[i]. price);
                         end;
end.
```

# Operations on structures

- Displaying (writing) data of a number of elements of structure type (car)

```
for i← 0 to (n -1) do
        begin
                write(List_car[i]. brand);
                write (List_car[i].c);
                 write(List_car[i]. price);
        end;
```

# Operations on structures

- Sort the list of cars according to their price (bubble sort)

```
var tmp : car; i,j:integer;
            …
            for i← (n-2) to 0 do
            begin
                        for j← 0 to i do
                        if(List_car[j].price> List_car[j+1].price) then
                        begin
                                    tmp← List_car[j];
                                     List_car[j]← List_car[j+1];
                                     List_car[j+1]←tmp;
                        end;
             end;
```