

Subscribe to DeepL Pro to edit this document.  
Visit [www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document) for more information.

# **Chapter 5**

# **Customised types**

This chapter is a continuation of the previous chapter on complex data types, in which we introduced the array type in its various formats: one-dimensional array, two-dimensional array (matrix) and character strings. In this chapter, we'll look at custom types.

1. **Structures (recordings)**

Tables contain elements of the same nature. However, data of different kinds may need to be grouped together. In a company's customer file, for example, we don't just store the customer's name, but also other data such as address, turnover, etc. These data are of different types, so theoretically we could create separate variables for these data. For example :

|  |  |
| --- | --- |
| **In algorithm** | **In C** |
| Variable  cp: integer  ca: real  name, town: character string | int cp ;  float ca ;  char name[20], city[20] ; |

But as data constitute a single entity, they all refer to a single person. In C, the type of this variable, which is made up of several elements of different types, is determined using **structures**.

Structure variables are data made up of a number of other objects, which are also variables. These partial variables that make up a structure are called structure **fields**.

* 1. **Structure declaration:** a structure is declared before the variables are declared.

|  |  |
| --- | --- |
| **In algorithm** | **In C** |
| customer **structure**  cp: integer  ca: real  name, town: character string  **End structure**  **variable**  ..... | struct client  {  int cp ;  float ca ;  char name[20], city[20] ;  } ;  ..... |

* 1. **Definition of the structured variable**

Once the structure has been declared, we can use it as shown in the following example:

|  |  |
| --- | --- |
| **In algorithm** | **In C** |
| customer **structure**  cp: integer  ca: real  name, town: character string  **End structure**  **variable**  c1: customer **structure** or  c1 : customer | struct client  {  int cp ;  float ca ;  char name[20], city[20] ;  } ;  **struct** client c1 ; |

**Remarks** :

1. The type of a field must not be that of the structure in which the field is contained.

|  |
| --- |
| struct client  {  int cp ;  float ca ;  char name[20], city[20] ;  Error  customer structure c ;  } ; |

1. It is permitted to take a structure of another type as a field after it has been declared, and this is known as **nesting structures**.

|  |
| --- |
| struct time  {  int hour ;  int minute ;  } ; |

The definition of another structure :

|  |
| --- |
| structured RDV  {  char name[20] ;  char jour[11]  struct time horaire ;  } ; |

1. The declaration of a structure and the definition of a variable of the type concerned can be condensed into the same instruction.

|  |
| --- |
| **In C** |
| struct client  {  int cp ;  float ca ;  char name[20], city[20] ;  } c1 ;🡪 variable of type struct client |

1. Several structured variables can be defined at once.

|  |  |
| --- | --- |
| **Example 1** | **Example 2** |
| struct client  {  int cp ;  float ca ;  char name[20], city[20] ;  } ;  struct client c1, c2, c3, ..., cn ; | struct client  {  int cp ;  float ca ;  char name[20], city[20] ;  } c1, c2, c3, ..., cn ; |

* 1. **Representation of a structure (record)**

Structures are made up of several data zones, corresponding to fields.

**Example:**

**C1**

|  |  |  |  |
| --- | --- | --- | --- |
| **c1.cp** | **c1.ca** | **c1.name** | **c1.city** |

**C1**

|  |  |  |  |
| --- | --- | --- | --- |
| **c2.cp** | **c2.ca** | **c2.name** | **c2.city** |

A structure is manipulated via its "fields". As with arrays, it is not possible to manipulate a structure globally, except to assign one structure to another of the same type.

For example: to display a structure, you need to display all its fields one by one.

Whereas the elements of an array can be accessed by their index, the fields of a structure can be accessed by their name, using the '**.**' operator.

Illustration: **structure\_name .field\_name**, represents the value stored in the structure field.

**Example**:

To access the name of variable c1, use the expression: "c1.name".

* 1. **Operations on structured variables**

1. **Assignment:** We need a specific operator here: the **.**

**The field operator** is placed between the name of the structured variable and that of the field concerned. The assignment syntax therefore takes the following form: <structured\_variable\_name>.<field.name>.

**Exp : client.name = 'zerrouk' ; client.city = 'Batna' ; client.cp = 05800 ; client.ca = 23.12 ; client.ca = 23.12+1 ; client.ca++ ;**

1. **Assignment between structure fields**: field values in a structure can be assigned to other fields in the same structured variable, or in other structured variables.

**Exp :** c1.ca = c2.ca ;

* 1. **Structure tables**

It is often the case that you want to process not just one record (structure) but several. For example, you want to be able to process a group of customers. So we won't create as many variables of the customer type as there are customers. Instead, we'll create a table containing all the customers in the group.

This is an array of structures (records).

struct client

{

int cp ;

float ca ;

char name[20], city[20] ;

} ;

|  |  |
| --- | --- |
| **In algorithm** | **In C** |
| **Var**  **T: array of [1..100] customer structures** | struct customer T[100] ; |

* Each element of the array is a structure, containing several variables of different types.
* A structure is accessed by its index in the table.
* **T[4]** represents the fourth customer in the table T
* **T[4].name** represents the name of the fourth client in the type table.

Member of another organisation

A structure can appear among the fields of another structure. In this case, it must be declared before the structure which contains it.

**Example**:

**Structure** Date

day, month, year: whole

**Finstructure**

customer **structure**

cp: integer

ca: real

name, town: character string

DN: **structure** Date

**Finstructure**

When a structure field is itself a structure, the field is accessed as follows: **variable\_structure**.**variable\_sub\_structure**.field.

**Examples**:

C1.DN.year← 2022 ; read( C1.DN.day, C1.DN.month, C1.DN.year) ; write(C1.DN.year)

In the case of an array, access is as follows: array\_name**[index]**.substructure\_variable.field

**Example**:

T[1].year← 2022 ; read( T[1].day, T[1].month, T[1].year) ;

write(T[1].DN.year)

1. **Enumerations (Definition of custom enum types)**

Enumerations allow a type to be defined by the list of values it can take. An enumeration object is defined by the keyword **enum** and a model identifier, followed by the list of values that the object can take: **enummodele** {constant-1, constant-2,...,constant-n};

In reality, objects of type **enum** are represented as int. The possible values constant-1, constant-2, ..., constant-n are encoded by integers from 0 to n-1. Some programs do not assign numerical values to common language objects such as continents, days of the week, names of months, etc., but instead create a type that is specifically textual and not numerical.

**Example:**

|  |  |
| --- | --- |
| **Algorithms** | **In C** |
| WEEK TYPE= (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)  COLOUR TYPE= (red, green, blue)  CONTINENT TYPE =(Africa, America, Asia, Europe, Australia) | **enum** WEEK  {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} ;  enumCOULEUR  {red, green, blue} ;  EnumCONTINENT  {Africa, America, Asia, Europe, Australia} ; |

**Then**, for example, the statement: enum WEEK days; defines a days variable, whose type is **enum WEEK**

**Rq**:

As with structures, with enumerated types you can combine the definition of a variable with the declaration of the type. As in the example below :

**enum** WEEK

{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} days ;

1. **Definition of custom types typedef**

To simplify program writing, you can assign a new identifier to a compound type using typedef :

**typedef** type synonym;

For example:

struct client

{

int cp ;

float ca ;

char name[20], city[20] ;

} ;

**typedef** struct client cl ;

main()

{

cl c1 ;

}

It's simpler than :

main()

{

struct client c1 ;

}

**Rq**:

It is possible to define a custom type at the same time as declaring the structure. The :

**typedefstruct** client

{

int cp ;

float ca ;

char name[20], city[20] ;

} cl;

Defines a structure type called **cl** which can be adopted as a synonym of the **struct client** type and not a variable with the name cl.

**Simple type redefinition**

You can rename simple types to draw attention to them, for example. **typedefint ENTIER ;** defines another name (synonym) **ENTIER** for the **int** type, to better highlight the nature of the variable defined.

Custom types are very often written in capitals, to distinguish them clearly from predefined types.

In the example :

char l1[80], l2[80], l3[80], l4[80], l5[80] ;

You can use the instruction: **typedef** char **LINE**[80]; to replace the previous instruction with the more concise: **LINE** l1, l2, l3, l4 ;

# **Exercises at home**

**Exercise 1**

Declare types that can store :

1. A student is identified by surname, first name, enrolment number, gender (F for female, M for male), age and grade average.
2. A group is defined by: its name, a list of its students and the number of students.
3. A section is defined by: its name, its groups (max 20), the number of groups, its year of study, its stream and its speciality.

**Exercise 2**

Let be a section of M groups (M≤20) where each group contains a maximum of 35 students. Write an algorithm that allows to:

* Enter the list of all students.
* Display the first and last name of the student(s) with the highest average in the section.
* Replace each first name Mohammed with Mohamed.

Count and display the number of students aged 20 and over.