Chapter 6 : Custom types

Dr. Abderrahmane Kefali

Senior Lecturer Class A,
Department of Computer Science,
University of May 8, 1945 - Guelma

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Introduction

The predefined types we've seen so far (integer, real, array...) do not allow us to describe all kinds of information encountered in real life. They do not, for example, allow us to group different types of information related to the same object into a single structure.

However, algorithmics and the majority of programming languages provide algorithm and program designers the ability to define new data types, known as ***custom types*** or ***user-defined*** types. These allow us to represent data structures composed of multiple elements of standard types.

The definition and manipulation of custom types are the focus of this chapter. More specifically, in this chapter, we are primarily interested in the ***record*** type.

2) Records (structures)

In the previous chapter, we saw that arrays allow us to gather several elements of the same type under a single name. However, in practice, we may also want to group information within the same structure that does not necessarily have the same type. To address this issue, algorithmics and programming languages have introduced new data structures called ***Records*** that are better adapted for representing this type of information.

2.1) Definition

A record also known as a structure, is a data structure that allows gathering within a single entity a set of data of the same type or different types associated with a single object.

The record is composed of a set of elements called ***fields***, where each field corresponds to a piece of data. Similar to the cells of an array, the fields of a record can be accessed individually for reading, writing, or manipulation.

2.2) Records in algorithmics

2.2.1)Declaration

Before declaring a record variable, you must first define its type in the declaration part of the algorithm using the keyword **TYPE**.

The general form of declaring a record type is as follows:

```
TYPE <name_Type > = RECORD
                     Begin
                    <name_Field1>: <type_Field1>;
                    <name_Field2>: <type_Field2>;
                    ..................
                    <name_Fieldn>: <type_Fieldn>;
                    End;
```

Where:

- **<name_Type>**: is the name of the defined record type.
- **<name_Field$_i$>**: is the name of the ith field of the record.
- **<type_Field$_i$>**: is the type associated with the ith field. It can be any simple type (integer, character, etc.) or structured type (array, etc.).

Once the type is defined, we can declare variables of this type as we normally would. The syntax for this declaration is as follows:

```
VAR <name_variable>: <name_type>;
```

Where **<name_variable>** is the name of the variable.

Examples:

1) The **Date** type is used to describe a real-world date, and it consists of three fields: **day**, **month**, and **year**, all of which are of type **Integer**.

```
TYPE Date = RECORD
            Begin
            Day: integer;
            Month: integer;
            Year: integer;
            End;
VAR D: Date;
```

2) The **Etudiant** type represents a university student. It is defined by his identification number (ID), last name, first name, age, and his marks in 9 courses.

```
TYPE Student = RECORD
               Begin
               ID,Age: integer;
               LastName,FirstName: String[20];
               Marks: Array[1..9] of real;
               End;
VAR Stud: Student;
```

2.2.2) Manipulating a Record

The only possible instruction for directly manipulating a variable of record type (in its entirety) without accessing its fields is assignment.

However, the fields of a record can be manipulated individually just like any other variable of a similar type.

a) Accessing a Field of a Record

We can access a field of the record by specifying the record's name followed by the field's name, with both separated by the dot operator (•). The syntax for access is as follows:

<name_Record>•<name_Field>

Examples:

- To access the **Day** field of the record **D**, we write: **D.Day**
- To access the **Age** field of the record **Stud**, we write: **Stud.Age**

b) Reading and Writing

To read a record, it is necessary to read each of its fields one by one. The same applies to displaying the record. The syntax for reading a field is as follows:

```
Read(<name_Record>.<name_Field>);
```

The syntax for writing a field is as follows:

```
Write(<name_Record>.<name_Field>);
```

Example:

```
Var D: Date:
```

Reading all the fields of the record **D** is done as follows:

```
Read(D.Day, D.Month, D.Year);
```

Similarly, displaying the record **D** is done as follows:

```
Write(D.Day, D.Month, D.Year);
```

Note:

It should be noted that unlike arrays, it is not possible to use a loop to read or write all the fields of a record.

c) Assignment

As mentioned earlier, assignment is possible between two record variables of the same type. Thus, the following form of instruction is accepted:

```
<name_Record1> ← <name_record2>;
```

This instruction implies that all the fields of the record **<name_record2>** are copied to the corresponding fields of the record **<name_record1>**.

We can also assign values to individual fields. To assign a value to a specific field of the record, we use the syntax:

```
<name_Record>.<name_Field> ← <value>;
```

The value and the field must be of the same type.

Exemple:

```
Var D1,D2: Date:
```

The following instructions are correct:

```
D1.Day ← 6;
D1.Month ← 12;
D1.Year ← 2022;
D2 ← D1;
```

d) The WITH...DO statement

To simplify access to the fields of a record, we can use a special statement: the **WITH** statement. Inside the **WITH** statement, we can directly manipulate the fields of the record without needing to add the record's name and a dot. As a result, instructions in the following form:

```
Read(<name_Record>.<name_Field1>);
<name_Record>.<name_Field2> ← <value>;
Write(<name_Record>.<name_Field3>);
```

Can be replaced, using the **WITH** structure, by:

```
WITH <name_Record> DO
        Begin
        Read(<name_Field1>);
        <name_Field2> ← <value>;
        Write(<name_Field3>);
        End;
```

Example:

Let the record **D** of type **Date** be as follows:

```
Var D: Date:
```

An example of accessing the fields of the record **D**, both without and with the **WITH** structure, is as follows:

Without the WITH structure

```
Read(D.Day);
D.Month ← 7;
D.Year ← 1992;
Write(D.Day,D.Month,D.Year);
```

➡

With the WITH structure

```
WITH D DO
    Begin
    Read(Day);
    Month ← 7;
    Year ← 1992;
    Write(Day,Month,Year);
    End;
```

2.2.3)Nesting of Records

A field within a record can itself be another record. This is referred to as the nesting of records.

Example:

```
TYPE Date = RECORD
            Begin
            Day, Month, Year: integer;
            End;
Student = RECORD
            Begin
            ID: integer;
            LastName, FirstName: String[20];
            Date_Birth: Date;
            Marks: Array[1..9] of real;
            End;
VAR Stud: Student;
```

To access the birth month of the student **Stud**, we write:

```
Stud.Date_Birth.Month
```

2.2.4)Arrays of Records

a) Declaration

To declare an array of records, you must first declare the type of the records it contains. Then, you declare the array type, and finally, you declare a variable of the defined array type.

Example:

The declaration of an array of 30 students, each defined by his ID number, last name, first name, age, and marks, is performed as follows:

```
CONST n = 30;
TYPE Student = RECORD
               Begin
               ID,Age: integer;
               LastName,FirstName: String[20];
               Marks: Array[1..9] of real;
               End;
     Tab = ARRAY[n] Of Student;
VAR T: Tab;
```

b) Accessing Fields of a Record in an Array

Accessing a field of a record within an array is done by specifying the name of the array, followed by the record number in square brackets, and then a dot, followed by the field name. The syntax for access is as follows:

```
<name_Array>[<index>].<name_Field>
```

Example:

T[4].FirstName refers to the **FirstName** field of the 5th record in the array **T**.

c) Manipulating an Array of Records

The fields of records in an array are manipulated separately in the same way described previously. However, traversing the records that make up the array's elements can be done using a loop.

Example:

Consider the following declaration:

```
CONST n = 30;
TYPE Student = RECORD
               Begin
               ID,Age: integer;
               LastName,FirstName: String[20];
               Marks: Array[1..9] of real;
               End;
     Tab = ARRAY[n] Of Student;
VAR T: Tab;i :Integer ;
```

The following instructions are accepted:

```
1)      Read(T[0].LastName);
2)      T[2].Marks[0] ← 18;
3)      Write(T[5].FirstName);
4)      T[10] ← T[9];
5)      For i ← 0 To n-1 Do
              Read(T[i].age);
```

2.3) Records in C language

In the C language, the term "structure" is often used rather than "record".

The declaration of a new structure template follows the following syntax:

2.3.1)Declaration of a Structure

The declaration of a structure variable can be done in various ways. However, in the C language, a structure is defined using the reserved keyword **`struct`**.

a) First declaration

It involves defining a structure template and listing the fields it contains, and then declaring a variable of the defined template.

The declaration of a new structure template follows the following syntax:

```
struct <name_Template>{
     <type_Fields1> <name_Fields1>;
     <type_ Fields2> <name_Fields2>;
     ...
     <type_ Fieldsn> <name_Fieldsn>;
};
```

Where: **`<name_Template>`** is the name of the defined template, **`<type_Fields`**$_i$**`>`** is the type of the i^{th} field of the structure, and **`<name_Fields`**$_i$**`>`** is the name of the i^{th} field of the structure.

Please note the mandatory semicolon at the end of a structure definition.

Next, to declare a variable of the structure type corresponding to the previous template, you use the following syntax:

```
struct <name_Template> <name_Variable>;
```

Where: **`<name_Variable>`** is the name of the declared variable.

Examples:

1) Definition of a structure template named **`Date`**, composed of 3 integer fields: **`Day`**, **`Month`**, **`Year`**, and the declaration of associated variables.

```
struct Date{
      int Day,Month,Year;
};
struct Date d1,d2;
```

Structure template labeled **`Student`** describing a university student known by their ID number, last name, first name, age, and marks. Then, declaring a structure variable following this template.

```
struct Student{
      int ID,Age;
      char LastName[20],FirstName[20];
      float Marks[9];
};
struct Student Stud;
```

Note:

Notice the necessity of repeating the **`struct`** keyword in the declaration of a structure variable following a template.

b) Declaration by Defining Type Synonyms

To avoid the repetition of the **`struct`** keyword with every declaration of a structure type variable, you can proceed with the second method of declaration.

This method involves the use of the **`typedef`** keyword to define what is called in the C language a ***shortcut*** or a ***type synonym***. Thus, using this keyword when defining the structure will allow to give a new name to this type.

Declaration by defining a type synonym follows the following syntax:

```
typedef struct {
     <type_Field1> <name_Field1>;
     <type_Field2> <name_Field2>;
     ...
     <type_Fieldn> <name_Fieldn>;
} <name_Type>;
```

Where: **`<name_Type>`** is the name of the defined type synonym, **`<type_Fieldi>`** is the type of the i^{th} field of the structure, and **`<name_Fieldi>`** is the name of the i^{th} field of the structure.

In this way, the declaration of a structure variable is done just like declaring a variable of any other type. The syntax for this declaration is as follows:

```
<name_Type> <name_Variable>;
```

Example:

The definition using **typedef** of the **Date** structure type is as follows:

```
typedef struct {
      int Day,Month,Year;
} Date;
```

The declaration of variables **D1** of **D2** of type **Date** is simply done as follows:

```
Date d1,d2;
```

2.3.2)Manipulating a Structure

Similar to in algorithmics, the manipulation of a structure is done through its fields. Furthermore, the only allowed operation on structures as a whole is assignment.

a) Accessing a Field of a Structure

Accessing a field follows the following syntax:

```
<name_Structure>.<name_Field>
```

Example:

Consider the following declaration:

```
struct Date d;
```

Ainsi:

d.Day refers to the **Day** field of the structure **d**.

b) Reading and Writing

Reading and writing a structure is done field by field.

The syntax for reading a field is as follows:

```
scanf("<format>",&<name_Structure>.<name_Field>);
```

The syntax for writing a field is as follows:

```
printf("<format>",<name_Structure>.<name_Field>);
```

Here, **<format>** represents the format for reading and writing the field ("**%d**", "**%f**", "**%s**",...).

Example:

Reading the fields of the structure **D** of type **Date** is done as follows:

```
scanf("%d%d%d",&D.Day,&D.Month,&D.Year);
```

Displaying a date stored in the variable **D** can be done as follows:

```
printf("%d/%d/%d",D.Day,D.Month,D.Year);
```

c) Assignment

Assignment between two structures allows to copy all the fields of the source structure to their corresponding fields in the target structure. This assignment follows the following syntax:

```
<name_Structure1> = <name_Structure2>;
```

Here, **<name_Structure1>** is the name of the target structure, and **<name_Structure2>** is the name of the source structure.

To assign a value to a specific field of the structure, you use the syntax:

```
<name_Structure>.<name_Field> = <value>;
```

Example:

Consider two variables **D1** and **D2** of type **Date**. The following instructions are accepted:

```
D1.Day = 6;
D1.Month = 12;
D1.Year = 2022;
D2 = D1;
```

Note:

In the C language, you can initialize a structure variable during its declaration, similar to arrays, using braces and commas. This initialization can be done in a sequential manner (the first value in the first field, the second value in the second field, and so on), selectively (specifying the fields along with their values), or in a mixed way (a combination of both).

Furthermore, it is possible to initialize only certain fields of the structure within the braces and leave the others empty.

Example:

Let's return to the definition of the **Date** type as defined earlier:

```
typedef struct {
      int Day,Month,Year;
} Date;
```

The following variables are initialized during their declaration:

```
Date birth_Date={7,2,2001};
```

Sequential initialization (**Day**, then **Month**, then **Year**).

```
Date entery_Date={.Month=10,.Year=2015,.Day=17};
```

Selective initialization.

```
Date exit_Date={20,.Year=2015};
```

Mixed initialization. The month is not initialized in this case.

2.3.3)Nesting of Structures

A structure field can itself be of a structure type, provided that this structure is defined before being used.

Example:

```
  typedef struct {
        int Day,Month,Year;
  } Date;
typedef struct{
        int ID;
        char LastName[20],FirstName[20];
        Date Date_Birth;
        float Marks[9];
} Studiant;
Studiant Stud;
```

To access the birth month of the student **Stud**, you need to use the dot operator "**.**" twice:

```
Stud.Date_Birth.Month
```

2.3.4)Arrays of structures

a) Declaration

Declaring an array of structures requires that the structure type has been declared beforehand. It's advisable to create a type synonym for the array type (using **typedef**) and declare variables of that type.

Example:

Declaring an array of 100 persons, each defined by his last name, first name, and age, is done as follows:

```
#define n 100
typedef struct{
        char LastName[20],FirstName[20];
        int age;
} Person;
typedef Person Tab_Pers[n];
Tab_Pers T;
```

b) Manipulation

Manipulation is the same as in algorithmics.

3) Other possibilities for defining types

In addition to records, there are other custom data types. We briefly mention some of them.

3.1) Enumerations

An enumeration, or an ***enumerated type***, is a type for which the designer explicitly lists an ordered set of possible values that a variable of this type can take. These values are explicitly defined and specified by identifiers (constants). The order of values is the order in which the identifiers were enumerated.

3.2) Interval type

This type allows us to define a range of values for a scalar type by specifying its lower and upper bounds. The types of the constants that serve as the bounds of the interval determine the scalar type from which the interval is derived. However, the interval can be a range of integer values or characters, but not real numbers or strings.

3.3) Set type

The set type defines an unordered collection of elements of the same type, and the number of elements is finite. You can perform classical mathematical operations and relations on sets, such as union, intersection, complement, equality, inclusion, and membership.