# Chapter 6: Records

***How can several data items of different types be grouped within the same structure? How can a real-world entity characterized by multiple attributes be represented in an algorithm? How can this information be stored and manipulated as a single logical variable while still allowing separate access to each data item? How can the clarity and organization of algorithms be improved when they handle complex data?***

**This chapter answers these questions.**

## 6.1 Introduction

In previous chapters, we studied basic data types (integer, real, character, Boolean) as well as arrays, which allow the storage of multiple values of the same type.
However, in many real situations, it is necessary to group together several pieces of information of different types describing the same entity (for example: a student, an employee, a medicine or a product).
To meet this need, records are used.

## 6.2 Concept of a Record

A record is a composite data structure consisting of a set of fields, where each field has its own name (identifier) and type.
All the fields together represent a single logical entity.

## 6.3 Declaration of a Record

The declaration of a record type in algorithmics is performed as follows:

## Syntax

**In Algorithmic:**

**Type** record_type_id = **Record**
  Field1 : Type1;
  Field2 : Type2;
  …
  FieldN : TypeN;
**End;**

**In C language:**

```
//Method 1
  struct record_id {
      Type1 Field1;
      Type2 Field2;
      ……..
      TypeN FieldN;
  };

//Method 2
   typedef struct {
       Type1 Field1;
       Type2 Field2;
       ……..
       TypeN FieldN;
   } record_type_id;
```

In Method 1, the record is declared only as a Tag[1], it is not a type however in Method 2 it is defined as a new type.

**Notes:**

- With typedef, the C compiler treats the declared type like a predefined type such as int or float.

- Each field corresponds to an elementary piece of information (A property), and all fields together form a new data type that can be used to declare variables.

## Example: Declaration of a record representing a student

[1] A tag is simply a name (label) given to a record/structure definition, not a data type by itself. It is mainly used to identify the structure so that it can be referred to later.

**In Algorithmics:**

**Type** Student = **Record**
  id : integer;
  lName, fNname : string [30];
  average : real;
**End;**

**In C language:**

```
//Method 1
struct Student {
    int id;
    char lName[30], fName[30];
    float average;
};

//Method 2
typedef struct {
    int id;
    char lName[30], fName[30];
    float average;
} Student;
```

Once a record type is defined, variables of this type can be declared and used like any other variable.

**Syntax:**

- **In Algorithmic**

**Variables** record_variable_id **:** record_type_id**;**

**Example:**
**Variables** s1, s2 : Student;

- **In C language:**

With the first method, we must use the keyword struct before declaring any record variable, for example `struct Student s1, s2;`

However in the second method we declare the variables directly without any keywords because here the record is defined as a type: `Student s1, s2;`

**Notes:**

- Each variable of a record type groups together all the fields defined in that type.

- Within a record, each field must have a unique identifier in order to avoid any ambiguity inside the structure. However, it is perfectly possible to reuse the same field identifier in different records, whether the associated type is the same or different, since each field is defined within the context of its own record.

**Example:**

```
Type Book = Record
   title : string [40];
   author : string [30];
    nbPage : integer;
End;

Type Movie = Record
   title : string [40];
   director : string [30];
End;
```

In this example, the field title appears in both records (Book and Movie). This does not cause any problem because each field belongs to a different record, even though they share the same identifier and type.

# 6.4 Accessing Record Fields

Access to a record field is performed using the dot operator (.). Each field can be read, modified, or displayed independently.

**Example:**

```
s1.id ← 12893;
s2.lName ← "BENADDA";
Read(s1.fName);
Write(s2.average);
```

# 6.5 Arrays of Records

In many applications, it is not sufficient to manipulate a single record. Very often, it is necessary to manage a collection of entities sharing the same structure, such as a list of students, employees, or products.
In such cases, an array of records is used.

An array of records allows storing multiple variables of the same record type, indexed like a standard array, while each element still contains all the fields of the record.

**Example: Array of students**

**In Algorithmic:**

```
Type Student = Record
   id : integer;
   lName, fName : string [30];
   average : real;
End;
```

**Variables** students : Array [1..100] of Student;

In this example, students is an array containing up to 100 student records. Each element of the array represents one student and can be accessed using its index.

**Syntax:** id_Array[Index].id_field ;

**Example:** Let's see how to access fields of an array of records g the previous example:

```
students[15].id ← 2023001;
Read(students[55].fName);
students[21].average ← 14.5;
Write(students[1].lName);
```

Thus, it is possible to manipulate both the array index and the fields of each record simultaneously, which makes arrays of records particularly suitable for managing structured collections of data.

# 6.6 Records Containing Other Records

A record may include fields that are themselves records. This makes it possible to represent entities composed of several structured attributes, and to model real-world data in a clear and hierarchical way.

**Example: Student record containing a Date of Birth record**

**In Algorithmics:**

```
Type DOB = Record
   day, month, year  : integer;
End;
```

**In C language:**

```
#include <stdio.h>
typedef struct {
    int day, month, year;
} DOB;
```

```
Type Student = Record
   id: integer;
   lName, fName : string [30];
   birth : DOB;
   average : real;
End;
```

```
typedef struct {
    int id;
    char lName[30], fName[30];
    DOB birth;
    float average;
} Student;
```

In this example, the *birth* field of the *Student* record is a record of type *DOB*, which groups the day, month, and year of birth into a single structured entity.

**Example: Accessing fields of a nested record assuming s as a variable of the Student record**

```
s.birth.day   ← 12;
s.birth.month ← 5;
s.birth.year  ← 2003;
```

The dot operator (.) is used repeatedly to access fields inside records.
This approach improves the clarity of algorithms by organizing related data into meaningful substructures, while still allowing direct access to each individual field.

# 6.7 The "With" Instruction (Statement)

In some programming languages, such as Pascal and Visual Basic, an instruction called with exists that allows programmers to avoid repeatedly rewriting the record variable identifier. However, this instruction does not exist in the C language; therefore, it will not be used in our algorithms.

**Example:**

```
s.id ← 2025773;
s.lName ← "BENADDA" ;
s.fName ← "Meriem";
s.average ← 15;
```

**Using the with instruction, we get**

```
With s do
Begin
  id ← 2025773;
  lName ← "BENADDA" ;
  fName ← "Meriem";
  average ← 15;
End;
```

# Conclusion

In this chapter, we introduced records as a structured data type that allows grouping related data of different types into a single entity. We showed how records improve clarity, organization, and data handling in algorithms