1.3 Records (Structures)

Many algorithmic problems involve objects described by several related values. Using separate variables for each value often leads to fragmented data representation and unclear algorithms. For example, an object described by two coordinates or by several attributes must be treated as a single logical entity.
Records provide a way to group related data, so that algorithms can manipulate them coherently.

1.3.1 Definition of a Record

A **record** (also called a structure) is a user-defined data type that groups several named components into one unit.

Each component:

- has its own type,
- has a specific role,
- belongs to a fixed position within the record.

A record represents a composite object, not a collection of independent values.

1.3.2 General Syntax of a Record

A record is defined using the key word `struct` by specifying its components:

```
struct RecordName {
    type component1;
    type component2;
    type component3;
    ...
};
```

Once defined, the record can be used as a new data type in algorithms.

Example: Point in a Plane

A point is described by two coordinates that must always be considered together.

```
struct Point {
    float x;
    float y;
};
```

Using this record ensures that the two coordinates are never separated in algorithmic operations such as distance computation, translation, or comparison.

Example : Time Representation

A time value can be described by three components:

```
struct Time {
    int hour;
    int minute;
    int second;
};
```

This representation is useful for algorithms that:

- compare times,
- compute durations,
- sort events chronologically.

1.3.3 Declaring and Using Record Variables

A variable of a record type can be declared as follows:

```
struct Point p;
```

Each component is accessed using the member selection operator (.):

```
p.x = 2.0;
p.y = 3.5;
```

This notation clearly indicates that x and y belong to the same object.

1.3.4 Records as Parameters and Results

Records can be used as:

- input parameters,
- output results,
- intermediate values.

Example: computing the midpoint of two points:

```
struct Point midpoint(struct Point p1, struct Point p2) {
    struct Point m;
    m.x = (p1.x + p2.x) / 2;
    m.y = (p1.y + p2.y) / 2;
    return m;
}
```

1.3.5 Arrays of Records

An array of records represents a sequence of structured elements.

```
struct Point polygon[50];
```

This structure is used in algorithms that process:

- sequences of positions,
- collections of values,
- ordered sets of objects.

Each element of the array is itself a complete record.

Example: Processing a Sequence of Records

```
struct Point centroid(struct Point pts[], int n) {
    struct Point c;
    c.x = 0.0f;
    c.y = 0.0f;
    int i;

    for (i = 0; i < n; i++) {
        c.x += pts[i].x;
        c.y += pts[i].y;
    }
    c.x /= n;
    c.y /= n;

    return c;
}
```

1.3.6 Nested Records

Records may contain other records as components.

Example:

```
struct Date {
    int day;
    int month;
    int year;
};

struct Event {
```

```
    struct Date date;
    int duration;
};
```

This allows complex data to be built in a structured and hierarchical manner.

1.3.7 Using typedef with Records

As algorithms grow, records are used more frequently:

- as parameters,
- as return values,
- inside arrays,
- inside other records.

Repeatedly writing the full record declaration makes algorithms harder to read and distracts from the logic of the algorithm itself.

To improve clarity, we introduce type definitions that allow records to be referenced using short and meaningful names.

A type definition assigns a new name to an existing record type as given by :

```
typedef struct {
    type field1;
    type field2;
} TypeName;
```

After this definition, `TypeName` can be used directly as a data type, without repeating the record keyword. This does not change the structure of the data; it only improves how algorithms are written and read.

Example :

Consider grouped scores used in an algorithm:

```
typedef struct {
    float test;
    float exam;
} Scores;
```

The algorithm using this data becomes very readable:

```
float finalScore(Scores s) {
    return 0.4 * s.test + 0.6 * s.exam;
}
```

1.3.8 Complete example

Consider an algorithm that manages the results of a student. Each student result is described by:

- an identification number,
- a continuous assessment score,
- a final exam score.

From these values, the algorithm must:

- compute the final score,
- decide whether the student passes or fails.

```
#include <stdio.h>

/* Enumeration for the decision outcome (finite set of states) */
typedef enum {
    FAILED,
    PASSED
} ResultStatus;

/* Record type for one student result (grouped data) */
typedef struct {
    int id;
    float continuous;
    float exam;
} StudentResult;

/* Classify the student */
ResultStatus classify(StudentResult s) {
    float finalScore;

      finalScore = 0.4f * s.continuous + 0.6f * s.exam;

      if (finalScore >= 10.0f)
        return PASSED;
    else
        return FAILED;
```

```
}

int main(void) {
    StudentResult student;


    printf("Enter student ID: ");
    scanf("%d", &student.id);

    printf("Enter continuous assessment score: ");
    scanf("%f", &student.continuous);

    printf("Enter final exam score: ");
    scanf("%f", &student.exam);


    ResultStatus st = classify(student);

    switch (st) {
        case PASSED: printf("Result: Passed");
                     break;
        case FAILED: printf("Result: Failed");
                     break;
    }


    return 0;
}
```