First Year Engineer in computer science

CHAPTER 12:

Structure – Union - Enumeration

2023-2024

Sid Ahmed BERRABAH

Structures

- We have already seen how an array allows us to designate a set of values of the same type under a single name, each of them being identified by an index.

- The **structure**, on the other hand, will allow us to designate a set of values of different types under a single name. Access to each element of the structure (called a **field**) will be done, this time, not by an indication of position, but by its name within the structure.

Structure Declaration

Declaration of a Structure Model

```
struct tag_name {
        data_type member1;
        data_type member2;
        // ... as many members as desired
};
```

struct: Keyword indicating the beginning of a structure declaration.

tag_name: Optional name for the structure type. This name allows you to create variables of this specific structure later.

data_type: Specifies the data type of each member variable within the structure. These can be basic types like int, float, char, or even other user-defined types like other structures.

member1, member2: Names given to each member (filed) variable within the structure. These names are used to access individual members of the structure.

Structure Declaration

Declaration of a Structure Model

- Example:

```
struct student {
        char name[100];
        char first_name[100];
        int age;
        char address[100];
 };
```

We have declared a data structure model with 4 fields (name, first name, age, and address). This model has the identifier student as its tag.

Note: The structure model declaration doesn't allocate any memory. It only defines the blueprint for the structure.

Structure Declaration

Declaration of a Structure variable

- To create variables of a structure type, you use the following syntax:

```
struct tag_name variable_name;
```

Example: The student structure model can be used to declare variables such as:

```
struct student s1, s2;
```

s1 and s2 are structure variables of type student.

Structure Declaration

- We can also declare variables of type structure without using a tag:

```
struct {
        char name[100];
        char first_name[100];
        int age;
        char address[100];
} e1, e2;
```

- Or, declare the tag and the variables at the same time:

```
struct student {
        char name[100];
        char first_name[100];
        int age;
        char address[100];
} e1, e2 ;
```

Manipulation of a Structure

- Each field of a structure can be manipulated like any variable of the corresponding type. The designation of a field is noted by following the name of the structure variable with a dot (.) followed by the name of the field defined in the model.

Example :

```
e1.age = 20 ;
scanf("%s", e2.name);
```

- It is possible to initialize the fields of a structure at the time of its declaration.

Example :

```
struct student e1 = {"Boukli", "Ahmed", 21, "5 rue de la liberté, 13000 Tlemcen"};
```

Manipulation of a Structure

- Unlike tables, it is possible to assign the contents of one structure to another structure globally, provided that they are both defined from the same model. For example, you can write:

```
e1 = e2 ;
```

- However, no global comparison operation is allowed.

Manipulation of a Structure

- We have already seen that variable declarations can be global (accessible to all functions) or local (accessible to only one function) depending on whether they are made in a function or outside of any function. This rule applies to both structures and structure models.

Manipulation of a Structure

- We have already seen that variable declarations can be global (accessible to all functions) or local (accessible to only one function) depending on whether they are made in a function or outside of any function. This rule applies to both structures and structure models.

```
struct global_model {
    /* ... */
};

struct global_model var_global;

int fct() {
    struct local_model {
            /* ... */
    };
    struct global_model var_local1;
    struct local_model var_local2;
    /* ... */
}
```

```
int main()
{
    struct global_model var_local3;
    /* ... */
}
```

In this example, we see that a structure can be local or global and that its model can also be local or global.

Array of Structures

- It is possible to create structure arrays, which are arrays whose elements are of a structure type.

  Example :

  ```
  struct student section[150] ;
  ```

section is an array of 150 elements of type student structure.

section[3] represents the structure of type student at the fourth element of the array section.

- You can access the fields of an element with:

  ```
  section[i].nom
  ```

Structures comportant d'autres structures

- A structure can contain other structures.

  Example : The structure date is used in the structure student which stores information about a student.

```
struct date {
        int day;
        int month;
        int year;
};
struct student {
        char name[100];
        char first_name[100];
        date birthday age;
        char address[100];
        date registration;
} e1, e2 ;
```

- The year of registration of the student e1 is accessed by:

  e1.registration.year

Passing a structure as a parameter to a function

- When you pass a structure to a function by value, the function gets a copy of the structure's data. Any changes that the function makes to the structure's data do not affect the original structure.

Passing a structure as a parameter to a function


```
struct str{
        int a;
        int b;
};

void swap(struct str x)
{
    printf("start swap : a = %d and b = %d\n", x.a, x.b);
    int save = x.a;
    x.a = x.b;
    x.b = save;
    printf("end swap : a = %d and b = %d\n", x.a, x.b);
}
int main()
{
    struct str x ={1,2};
    printf("before swap call : a = %d and b = %d\n", x.a, x.b);
    swap(x);
    printf("after swap call : a = %d and b = %d\n", x.a, x.b);
    return 0;
}
```

Note that the str structure model must be global to be used in the swap function.

Passing a structure as a parameter to a function


```c
struct str{
        int a;
        int b;
};

void swap(struct str x)
{
    printf("start swap : a = %d and b = %d\n", x.a, x.b);
    int save = x.a;
    x.a = x.b;
    x.b = save;
    printf("end swap : a = %d and b = %d\n", x.a, x.b);
}
int main()
{
    struct str x ={1,2};
    printf("before swap call : a = %d and b = %d\n", x.a, x.b);
    swap(x);
    printf("after swap call : a = %d and b = %d\n", x.a, x.b);
    return 0;
}
```

Note that the str structure model must be global to be used in the swap function.

Output :

```
before swap call : a = 1 and b = 2
start swap : a = 1 and b = 2
end swap : a = 2 and b = 1
after swap call : a = 1 and b = 2
```

Passing a structure as a parameter to a function

As we have already seen for simple variables or arrays, in order to be able to modify the structure in the function, it is the address of the structure that must be passed to the function.

```
void swap(struct str * x)
{
   printf("start swap : a = %d and b = %d\n", (*x).a, (*x).b);
   int save = (*x).a;
   (*x).a = (*x).b;
   (*x).b = save;
   printf("end swap : a = %d and b = %d\n", (*x).a, (*x).b);
}
```

Passing a structure as a parameter to a function

As we have already seen for simple variables or arrays, in order to be able to modify the structure in the function, it is the address of the structure that must be passed to the function.

```
int main()
{
    struct str x ={1,2};
    printf("before swap call : a = %d and b = %d\n", x.a, x.b);
    swap(&x);
    printf("after swap call : a = %d and b = %d\n", x.a, x.b);
    return 0;
}
```

Output :

```
before swap call : a = 1 and b = 2
start swap : a = 1 and b = 2
end swap : a = 2 and b = 1
after swap call : a = 2 and b = 1
```

Passing a structure as a parameter to a function

- In the swap() function, a pointer to the structure is now used to collect the address of the parameter. Access to the fields of the structure is done using (*z).a and (*z).b.

- *z is the structure pointed to by z.

- **The parentheses are important because the * operator has lower precedence than the (.) operator.**

- To avoid this somewhat cumbersome notation, the C language offers a facility, the -> operator. Access to the field is then done with: z->a and z->b.

Passing a structure as a parameter to a function

- The swap function can be rewritten as follows:

```
void swap(struct str * x)
{
    printf("start swap : a = %d and b = %d\n", x->a, x->b);
    int save = x->a;
    x->a = x->b;
    x->b = save;
    printf("end swap : a = %d and b = %d\n",x->a, x->b);
}
```

Exercises

- **Exercise 1:**

Write a C program that defines a point structure that will contain the two coordinates of a point on the plane. Then it reads two points and displays the distance between these two points.

- **Exercise 2:**

Write a C program that reads the times made by two runners (runner1 and runner2) and displays the winner who made the minimum time. The time is made up of minutes and seconds.

- **Exercise 3:**

Write a C program that reads the times made by N runners and classifies them by order of arrival. Use functions to: read the names of the runners and their times; a function to compare the times made by two runners; a function to classify the runners; and a function to display the result.

Exercises

◆ **Exercise 1:** Write a C program that defines a point structure that will contain the two coordinates of a point on the plane. Then it reads two points and displays the distance between these two points.

```c
struct Point{
        int x;
        int y;
};

int main()
{
   struct Point p1, p2;
   printf("give the x-coordinate of P1 : "); scanf("%d", &p1.x);
   printf("give the y-coordinate of P1 : "); scanf("%d", &p1.y);

   printf("give the x-coordinate of P2 : "); scanf("%d", &p2.x);
   printf("give the y-coordinate of P2 : "); scanf("%d", &p2.y);

   double d = sqrt((p2.x-p1.x)*(p2.x-p1.x) + (p2.y-p1.y)*(p2.y-p1.y));

   printf("the distance between p1(%d,%d) and p2(%d,%d) is %f\n",p1.x,p1.y,p2.x,p2.y, d);
   return 0;
}
```

◆ **Exercise 2:** Write a C program that reads the times made by two runners (runner1 and runner2) and displays the winner who made the minimum time. The time is made up of minutes and seconds.

```c
struct Runner{
    char name[50];
    int minutes;
    int seconds;
};

int main()
{
   struct Runner r1, r2;
   printf("give the name of runner 1 : "); scanf("%s", r1.name);
   printf("give the minutes made by %s : ",r1.name); scanf("%d", &r1.minutes);
   printf("give the seconds made by %s : ",r1.name); scanf("%d", &r1.seconds);

   printf("give the name of runner 2 : "); scanf("%s", r2.name);
   printf("give the minutes made by %s : ",r2.name); scanf("%d", &r2.minutes);
   printf("give the seconds made by %s : ",r2.name); scanf("%d", &r2.seconds);

   int t1 = r1.minutes*60+r1.seconds;
   int t2 = r2.minutes*60+r2.seconds;

   if (t1<t2) printf("%s(%dm:%ds) is the winner",r1.name,r1.minutes,r1.seconds);
   else printf("%s(%dm:%ds) is the winner",r2.name,r2.minutes,r2.seconds);
   return 0;
}
```

◆**Exercise 3:** Write a C program that reads the times made by N runners and classifies them by order of arrival. Use functions to: read the names of the runners and their times; a function to compare the times made by two runners; a function to classify the runners; and a function to display the result.

```c
typedef struct {
   char name[100];
   int minutes;
   int seconds;
}RUNNER;

void readRunners(RUNNER *T, int N)
{
   for(int i = 0; i < N; i++){
       printf("give the name of the runner %d : ", i); scanf("%s", T[i].name);
       printf("give the minutes made by runner %d : ", i); scanf("%d", &T[i].minutes);
       printf("give the seconds made by runner %d : ", i); scanf("%d", &T[i].seconds);
   }
}

void displayRunners(RUNNER *T, int n)
{
   RUNNER *p;
   for(p= T; p < T+n; p++)
       printf("%s made %dm:%ds\n", p->name, p->minutes, p->seconds);
}
```

```c
int toseconds(RUNNER r)
{
   return r.minutes*60 + r.seconds;
}

int compare2runners(RUNNER r1, RUNNER r2)
{
   int t1= toseconds(r1);
   int t2= toseconds(r2);
   if (t1>t2) return 1;
   else return 0;
}

void sortRunners(RUNNER *T, int N)
{
   int i,j;
   RUNNER save;
   for(i= 0; i < N; i++)
       for(j=0; j<N-1-i; j++)
           if(compare2runners(T[j],T[j+1])){
               save = T[j];
               T[j] = T[j+1];
               T[j+1] = save;
           }
}

int main()
{
   RUNNER *Runners;
   int N;
   printf("give the number of runners :"); scanf("%d", &N
   Runners = (RUNNER *)malloc(N*sizeof(RUNNER));

   readRunners(Runners,N);
   sortRunners(Runners, N);
   displayRunners(Runners, N);
   return 0;
}
```

New types definition

In C, it is possible to define types that do not exist from simple types using the typedef keyword.

Example of using typedef:

```
typedef  unsigned long int * PULONG
```

keyword — declaration — name of the new type

The new type can then be used in a variable declaration:

```
PULONG x;
```

x is of type PULONG, which is a pointer to an unsigned long integer.

New types definition

The **typedef** operator does not create a new data type. It only assigns a new name (alias) to an existing type.

Another example:

```
typedef int tab[10];
```

This allows us to declare a variable of type tab, which is equivalent to declaring an array of 10 integers:

```
tab y;
```

New types definition

The typedef operator can also be used to create aliases for other types, such as structures:

```
typedef struct {
        int x;
        int y;
} POINT;

POINT *z ;
```

The new type POINT is a structure with two integer fields. This new type can be used to declare variables

The z variable is a pointer to a POINT structure. The * symbol before the z variable indicates that it is a pointer.

New types definition

We can declare multiple new types in the same typedef:

```
typedef struct {
        char name[100];
        char first_name[100];
        int age;
} STUDENT, *PSTUDENT, ArrSTUDENT[100] ;

STUDENT s ;  // s is a variable of type structure
PSTUDENT *p ;  // p is variable of type pointer to structure
ArrSTUDENT T ; // T is an array of 100 structures.
```

Exercise 4

Write a C program to calculate the working hours of an employee. The employee is defined by their name, arrival time (hour:minute:second), departure time (hour:minute:second), and working time (hour:minute:second).

Exercise 4

Write a C program to calculate the working hours of an employee. The employee is defined by their name, arrival time (hour:minute:second), departure time (hour:minute:second), and working time (hour:minute:second).

```
typedef struct {
    int Hour;
    int minutes;
    int seconds;
} TIME;
typedef struct{
    char name[50];
    TIME arrival;
    TIME departure;
    TIME working;
} EMPLOYEE;
```

```c
int TimeToSeconds(TIME t)
{
    return t.Hour*3600 + t.minutes*60 + t.seconds;
}
TIME SecondsToTime(int s)
{
    TIME t;
    t.Hour = s / 3600;
    s %= 3600;
    t.minutes = s / 60;
    t.seconds = s % 60;
    return t;
}
void stayAtWork(EMPLOYEE *e)
{
    int seconds = TimeToSeconds(e->departure) - TimeToSeconds(e->arrival);
    e->working = SecondsToTime(seconds);
}
int main()
{
   EMPLOYEE e = {"Ahmed",{7,59,10},{16,5,0},{0,0,0}};
   stayAtWork(&e);
   printf("%s arrived at %d:%d:%d and left at %d:%d:%d, he stayed at work %d:%d:%d",
          e.name, e.arrival.Hour,e.arrival.minutes, e.arrival.seconds,
          e.departure.Hour, e.departure.minutes, e.departure.seconds,
          e.working.Hour, e.working.minutes, e.working.seconds);
   return 0;
}
```

output

```
Ahmed arrived at 7:59:10 and left at 16:5:0, he stayed at work 8:5:50
```

Exercise 5

Assuming the following sizes for basic data types:

Type	Size (bytes)
`char`, `unsigned char`	1
`short int`, `unsigned short int`	2
`int`, `unsigned int`, `long int`, `unsigned long int`	4
`float`	4
`double`, `long double`	8
Pointer of any type	4

Exercise 5 - suite

/Data structures to store information about the cells of the network to which a mobile phone is connected:

```
#define SIZE 100
/* Cell Information */
struct cell_information
{
    char name[SIZE];                    //  Cell name
    unsigned int identifier;            // Cell identifier
    float signal_quality;               // Signal Quality (between 0 and 100)
    struct information_carrier *carrier_ptr;        / pointer to the second structure
};

/*  Carrier information */
struct information_carrier
{
    char name[SIZE];                    // Carrier name
    unsigned int priority;              // Connection priority
    unsigned int last_checked;          // Last checked
};
```

Exercise 5 - suite

Questions :

1. What is the size in bytes of a data of type struct cell_information?

2. If a variable of type struct cell_information is stored in memory starting at position 100, what will be the addresses of each of its fields?

3. Which of the following two variables takes up more space in memory::

```
struct cell_information a;
struct cell_information *b;
```

4. What is the size of the following variables:

```
struct cell_information *ptr1, *ptr2;
struct information_carrier *i1, *i2;
```

Exercise 6: Given the following code:

```c
#include <stdio.h>
#include <stdlib.h>

struct pack3
{
    int a;
};
struct pack2
{
    int b;
    struct pack3 *next;
};
struct pack1
{
    int c;
    struct pack2 *next;
};
```

```c
int main()
{
    struct pack1 data1, *data_ptr;
    struct pack2 data2;
    struct pack3 data3;

    data1.c = 30;
    data2.b = 20;
    data3.a = 10;
    data_ptr = &data1;
    data1.next = &data2;
    data2.next = &data3;

    printf("%d\n", data1.c);
    printf("%d\n", data_ptr->c);
    printf("%d\n",data_ptr.c);
    printf("%d\n",data1.next->b);
    printf("%d\n",data_ptr->next->b);
    printf("%d\n",data_ptr.next.b);
    printf("%d\n",data_ptr->next.b);
    printf("%d\n",(*(data_ptr->next)).b);
    printf("%d\n",data1.next->next->a);
    printf("%d\n",data_ptr->next->next.a);
    printf("%d\n",data_ptr->next->next->a);
    printf("%d\n",data_ptr->next->a);
    printf("%d\n",data_ptr->next->next->b);
}
```

Exercise 6:

Decide which of the following expressions are correct, and if so, which value is being accessed.

Expression	Correct	Value
`data1.c`		
`data_ptr->c`		
`data_ptr.c`		
`data1.next->b`		
`data_ptr->next->b`		
`data_ptr.next.b`		
`data_ptr->next.b`		
`(*(data_ptr->next)).b`		
`data1.next->next->a`		
`data_ptr->next->next.a`		
`data_ptr->next->next->a`		
`data_ptr->next->a`		
`data_ptr->next->next->b`		