Lecture 7
Structures-Part 1

Dr. Hacer Yalım Keleş

Programming in ANSI C, Kumar

Structures and Unions

we introduced arrays that provide the facility for grouping related data items of the same type into a single object.

at times we need to group related data items of different types.

C provides a facility, called *structures,* that allows a fixed number of data items, possibly of different types, to be treated as a single object.

BASICS OF STRUCTURES

A *structure* is a collection of logically related data items grouped together under a single name, called a *structure tag*. The data items that make up a structure are called its *members, components,* or *fields,* and can be of different types. The general format for defining a structure is

```
struct tag
   {
     variable declarations
   };
```

For example, the structure for the inventory record of a stock item may be defined as

```
struct item
  {
    int itemno;
    float price;
    float quantity;
    float eoq;
    int reorderlevel;
  };
```

All the members of a structure can be of the same type, as in the following definition of the structure `date`:

```
struct date
  {
    int day, month, year;
  };
```

Structure Variables

A structure definition defines a new type, and variables of this type can be declared by including a list of variable names between the right brace and the terminating semicolon in the structure definition. For example, the declaration

```
struct date
  {
    int day, month, year;
  } order_date, arrival_date;
```

declares `order_date` and `arrival_date` to be variables of type `struct date`.

The structure tag can be thought of as the name of the type introduced by the structure definition, and variables can also be declared to be of a particular structure type by a declaration of the form:

```
struct tag variable-list;
```

For example, the declarations

```
struct item m68020;
struct item intel386;
```

or the single declaration

```
struct item m68020, intel386;
```

declares m68020 and intel386 to be variables of type struct item.

The structure tag may be omitted if all the variables of a particular structure type have been declared when the structure is defined. Thus, the declaration

```
struct
  {
    float r, theta;
  } polar1, polar2;
```

declares `polar1` and `polar2` to be structure variables of the same type whose member variables are the two floating-point variables, `r` and `theta`. However, in such a case, we cannot subsequently declare another variable `polar3` whose type is the same as that of `polar1` and `polar2`.

Each occurrence of a structure definition introduces a new structure type that is neither the same nor equivalent to any other type. Thus, given that

```
struct { char c; int i;  } u;
struct { char c; int i;  } v;
struct si { char c; int  i; } w;
struct s2 { char c; int  i; } x;
struct s2 y;
```

the types of `u`, `v`, `w`, and `x` are all different,

but the types of `x` and `y` are the same.

NOTE THAT a structure definition does not allocate any storage; it merely describes a template or the shape of the structure. Storage is allocated only when a variable of the corresponding type is declared.

Structure Initialization

The declaration

```
struct date
  {
    int day, month, year;
  } independence = {15, 8, 1947};
```

initializes the member variables `day`, `month`, and `year` of the structure variable `independence` to `15`, `8`, and `1947` respectively, and the declaration

```
struct date republic = {26, 1, 1950};
```

initializes the member variables `day`, `month`, and `year` of the structure variable `republic` to `26`, `1`, and `1950` respectively.

What happens if we provide fewer initializers than there are member variables in the structure?

If there are fewer initializers than there are member variables in the structure, the remaining member variables are initialized to zero. Thus, the initialization

```
struct date newyear = {1, 1};
```

is the same as

```
struct date newyear = {1, 1, 0};
```

It is an error to provide more initializers than the number of member variables.

Accessing Structure Members

C provides a special operator, the *structure member* or *dot* operator, to access the individual members of a structure variable by a construction of the form

structure-variable. member-name

Thus, the statements

```
struct date man_on_moon;
man_on_moon.day = 20;
man_on_moon.month = 7;
man_on_moon.year =1969;
```

set the values of the member variables `day`, `month`, and `year` within the variable `man_on_moon` to `20`, `7`, and `1969` respectively

```
struct date today;
 ...
if (today.day == 25 && today.month == 12)
   printf("merry Christmas");
```

tests the values of **day** and **month** to check if they are 25 and **12** respectively, and if so, prints **merry Christmas**.

Structure Assignment

A structure variable may be assigned to another structure variable of the same type. Thus, given that

```
struct date american, revolution = {4, 7, 1776};
```

the assignment

```
american = revolution;
```

assigns `4` to `american.day`, `7` to `american .month`, and `1776` to `ameri-can. year.`

```c
#include <stdio.h>

/* define a structure type struct launchdate */
/* declare satellite to be a variable of this type */
struct launchdate {int day, month, year;} satellite;

int main(void)
  {
    /* declare sputnik2 to be of type struct launchdate
       and initialize it */
    struct launchdate sputnik2 = {3, 11, 1957};

    /* assign sputnik2 to satellite */
    satellite = sputnik2;

    /* access members of satellite */
    printf("Laika went orbiting on %d-%d-%d\n",
        satellite.month, satellite.day, satellite.year);

    return 0;
  }
```

Size of a Structure

The size of a structure can be determined using the `sizeof` operator discussed in Section 7.8.1. For example, given the declaration

```
struct porous
  {
    char c;
    long l;
  };
```

the statement

```
i = sizeof(porous);
```

assigns to `i` the size of the structure `porous` in bytes.

The size of a structure need not be the same as the sum of the sizes of its members; there may be holes in structures due to the alignment requirements for different objects. For example, on most machines, the size of `porous` is likely to be 8, rather than 5.

Nested Structures

Structures can be nested. For example, the structures `item` and `date` may be embedded inside another structure that records the ordering date and the quantity of an item to be purchased as in

```
struct order
  {
    struct item purchaseitem;
    struct date orderdate;
    float quantity;
  };
```

Note that the member variable `quantity` in the structure `order` does not conflict with the member variable `quantity` in the structure `item` embedded in it.

Another example of a nested structure is the definition of the following structure that contains two occurrences of the structure `date`:

```
struct project
  {
    int number;
    struct date start;
    struct date finish;
    float budget;
  };
```

There is no limit on the depth of nesting. For example, we can have

```
struct projects
   {
     struct date as_of;
     float revenue;
     struct project starbase;
     struct project hal;
   };
```

However, a structure cannot be nested within itself. Thus, the following is illegal:

```
struct company
  {
    struct projects government;
    struct projects industrial;
    struct company parent;          /* illegal */
  };
```

```
struct stockindex
  {
    float high, low, close;
    struct date weekending;
  } dowjones =
    {2520.79, 2381.19, 2520.79, {19, 10, 1990}};
```

The inner pair of braces is optional and certainly not necessary when all the initializers are present. Thus, we can have

```
struct stockindex sp500 =
    (312.48, 296.41, 312.48, 19, 10, 1990};
```

```
struct projects
  {
    struct date as_of;
    float revenue;
    struct project starbase;
    struct project hal;
  };
```

Here is a somewhat more involved example:

```
struct projects space =
  {
    {31, 12, 2000},      /* as_of */
    1000,                /* revenue */
    {                    /* starbase */
        007,                 /* number */
        {1, 6, 1999},        /* start */
        {1, 6, 2001},        /* finish */
        100                  /* budget */
    },
    {013, {1, 1, 1999}, {1, 1, 2001}, 500} /* hal */
  };
```

A particular member inside a nested structure can be accessed by repeatedly applying the dot operator. Thus, the statement

```
space.hal.budget = 200;
```

```
printf("%d", space.starbase.start.year);
```

This statement prints?

```
struct date newdate = {31, 12, 2001};
spaoe.hal.finish = newdate;
```

resets the completion date for the `hal` project to December 31, 2001.

Pointers to Structures

A pointer to a structure identifies its address in memory, and is created in the same way that a pointer to a simple data type such as `int` or `char` is created. For example, the declaration

```
struct date carnival, *mardigras;
```

declares `carnival` to be a variable of type `struct date`, and the variable `mardigras` to be a pointer to a `struct date` variable. The address operator & is applied to a structure variable to obtain its address. Thus, the assignment

```
mardigras = &carnival;
```

makes `mardigras` point to `carnival`.

The pointer variable `mardigras` can now be used to access the member variables of `carnival` using the dot operator as

```
(*mardigras).day
(*mardigras).month
(*mardigras).year
```

The parentheses are necessary because the precedence of the dot operator (.) is higher than that of the dereferencing operator (*). In the absence of the parentheses, the preceding expressions are interpreted as

```
*(mardigras.day)
*(mardigras.month)
*(mardigras.year)
```

Pointers are so commonly used with structures that C provides a special operator ->, called the *structure pointer* or *arrow* operator, for accessing members of a structure variable pointed to by a pointer. The general form for the use of the operator -> is

pointer-mme->member-mme

Thus, the preceding expressions for accessing member variables of the structure pointed to by `mardigras`, written using the dot operator/can equivalently be written using the arrow operator as

```
mardigras->day
mardigras->month
mardigras->year
```

It is permissible to take addresses of the member variables of a structure variable; that is, it is possible for a pointer to point into the middle of a structure. For example, the statement

```
float *sales = &space.revenue;
```

```
struct date *completion = &space.starbase.finish;
```

makes `completion` point to the structure member `finish` within `starbase` within `space`. Therefore, the statement

```
printf("year = %d sales = %f", *year, *sales);
```

prints

```
year = 2000 sales = 1000.000000
```

whereas the statement

```
printf("starbase completion day = %d", completion->day);
```

prints

```
starbase completion day = 1
```

The arrow operator has the same precedence as the dot operator, and is left-associative.

Pointers to structures improve execution speed; the larger the structure, the greater is the gain because transferring pointers from one part of the program to another is usually faster than copying all the individual components of a large structure. A major use of pointers to structures, as we shall see in the next section, is in the interaction of structures with functions.