Lecture 7
Structures-Part 3

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, by Kumar

Arrays of Structures

```
struct item motor[1000];
```

This statement declares `motor` to be an array containing 1000 elements of the type `struct item`.

Arrays of structures can be initialized in a manner similar to the initialization of multidimensional arrays as in

```
struct date birthdays[3] =
   {
      {14, 3, 1879},     /* Einstein */
      {0, 10}                    /* Gandhi */
   };
```

or equivalently as

```
struct date birthdays[3] =
   {
      {14, 3, 1879},
      {0, 10, 0},
      {0, 0, 0}
   };
```

The inner pair of braces is optional and can be omitted when all the initializers are present, as in

```
struct date atombomb[2] =
  {
    6, 8, 1945,        /* hiroshima */
    9, 8, 1945         /* nagasaki .*/
  };
```

A particular member variable inside an array of structures can be accessed using the array subscript and dot operators. Thus, the statement

```
birthdays[1].day = 2;
birthdays[2] = birthdays[0];
```

Pointers can also be used to access structure elements and member variables thereof. Thus, the statement

```
struct date *bday = &birthdays[1];
bday->day =2;
*(bday +1) = *(bday -1);
```

```
struct ndate
   {
      int day;
      char weekday[10];
      int month;
      char monthname[10];
      int year;
   };
```

A structure variable of the type ndate can now be declared as

```
struct ndate newcentury;
```

and initialized at the same time as

```
struct ndate newcentury =
      {1, {'m','o','n','d','a','y','\0' }, 1,
            {'j','a','n','u','a','r','y','\0'}, 2001);
```

or equivalently as

```
struct ndate newcentury =
      {1, "monday", 1, "january", 2001};
```

```
struct ndate newcentury =
        {1, "monday", 1, "january", 2001};
```

An element of an array contained in a structure can be accessed using the dot and array subscript operators. Thus, the statement

```
printf("%c", newcentury.monthname[2]);
```

prints

n

```
struct time
  {
    int val [3];
  } noon = {12, 0, 0};
```

and

```
void advance(struct time tm)
  {
    int i;

    for (i =0; i < 3; i++) tm.val[i] += 5;
  }
```

the statements

```
advance(noon);
for (i =0; i < 3; i++) printf("%d ", noon.val[i]);
```

print ?

```
advance(noon);
for (i =0; i < 3; i++) printf("%d ", noon.val[i]);
```

print

```
12 0 0
```

since the structure variable `noon` is passed by value and changes to the array member variable `val` inside `advance` are not reflected in the argument `noon`.

Arrays of Structures Containing Arrays

A natural corollary of the discussion so far is that we can define arrays of structures that contain arrays as member variables. Thus, we can have

```
struct student
  {
   char name[10];
    float height;
    int grades[4];
  } students[3] =
      {
        { "sleepy", 9, {1, 0, 0, 1} },
        { "happy", 7, {4, 3, 4, 4} },
        { "dopey", 8, {2, 2, 1, 1} }
      };
```

Elements of the member arrays can be accessed as before using the array subscript and dot operators. Thus, the statement

```
printf("%s  %d",  students[1].name,  students[1].grades[1]);
```

prints ?

Pointers can also be used to access the elements of the member arrays. Thus, the preceding `printf` can also be written as

```
struct student *sp = &students[1];
printf("%s %d", sp->name, sp->grades[1]);
```

or as

```
printf ("%s %d", sp->name, *(sp->grades + 1));
```