Lecture 7
Structures-Part 2

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

STRUCTURES AND FUNCTIONS

Scope of a Structure Type Definition

The scoping rules for a structure type definition are identical to those discussed for variable names in Chapter 5. A structure may be defined within a function or outside of any function; in the former case, it is called the *local,* and in the latter, the *external* structure definition.

a local structure definition permits only local variables of that structure type to be defined within the same block.

```
/* external structure definition */
struct rate {float fromdollar, todollar;};

/ * external structure variable * /
struct rate yen = {.007868, 127.10};

void germany(void)
  {
    struct rate mark = {.6760, 1.4793};      /* legal */
    struct date                             /* local structure definition */
        {int day, month, yr;}

    struct date on = {21, 11, 90};          / * local structure variable */


     printf("dollar = %f mark\n",
              mark. fromdollar) ;           /* legal */
    printf("on %d-%d-%d\n",
              on.month, on.day, on.yr);   /* legal */
  }


 void japan(void)
   {
     struct date today;        /* illegal; the definition of
                                     date  unavailable */


     printf("dollar = %f yen\n", yen.fromdollar); /* legal */
   }
```

Structure definitions are usually collected in a header file. This file is then included in those modules that need these structure definitions.

```
struct point
{
    float x, y;
  };

struct circle
  {
    float r;          /* radius */
    struct point o;   /* center */
  };


float sqr(float x)
  {
    return x * x;
  }

int contains (float cr, float cx, float cy,
              float px, float py)
  {
    return sqr(cx - px) + sqr(cy - py) > sqr(cr) ? 0 : 1;

  }
```

```
struct circle c = {2, {1, 1}};
struct point .p = {2, 2}; .
```

the function contains can be called as

```
contains (c.r, c.o.x, c.o.y, p.x, p.y)
```

What is the return value?

```
struct circle c = {2, {1, 1}};
struct point .p = {2, 2}; .
```

the function contains can be called as

```
contains (c.r, c.o.x, c.o.y, p.x, p.y)
```

and it will return 1 (true), since the distance of the point (2,2) from the center of the circle (1,1) is less than 2, the radius of the circle.

The second method involves passing the complete structure to a function

```
int contains(struct circle c, struct point p)
  {
    return sqr(c.o.x - p.x) + sqr(c.o.y - p.y)
               > sqr(c.r) ? 0 : 1;
  }
```

and called simply as

```
contains(c, p)
```

NOTE THAT:

Unlike array names, structure names are not pointers and are passed by value. Thus, when a structure name is provided as argument, the entire structure is copied to the called function, and changes to member variables of the structure argument in the called function are not reflected in the corresponding structure variable in the calling function.

The third method involves passing pointers to the structure variables as the function arguments.

```
int contains (struct circle *c, struct point *p)
    {
      return sqr(c->o.x - p->x)+sqr(c->o.y - p->y)
                > sqr(c->r) ? 0 : 1;
    }
```

and called as

```
contains(&c, &p)
```

NOTE THAT when a called function is provided with the address of a structure variable supplied as argument, any change in the called function to the member variables accessed using this address will be reflected in the structure variable in the calling function.

This method becomes particularly attractive when large structures have to be passed as function arguments because it avoids copying overhead.

Structures as Function Values

Structures may be returned as function values. Reconsider, for example, the problem of converting rectangular coordinates of a point into polar coordinates

```
struct polar convert(struct rectangular rec);
```

that takes as argument a structure of the type

```
struct rectangular
  {
    float x, y;
  };
```

giving the rectangular coordinates of a point, and returns a structure of the type

```
struct polar
  {
    float r, theta;
  };
```

giving the polar coordinates of the point.

```c
#include <math.h>

struct polar convert(struct rectangular rec)
  {
    struct polar pol;

    if (rec.x == 0 && rec.y == 0) /* origin */
        pol.r = pol.theta = 0;
    else
      {
        pol.r = sqrt(rec.x * rec.x + rec.y * rec.y);
        pol.theta = atan2(rec.y, rec.x);
      }

   return pol;
  }
```

the program fragment

```
struct rectangular r = {2, 1};
struct polar p;

p = convert(r);
printf("%f %f", p.r, p.theta);
```

prints

```
2.236068 0.463648
```

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

struct polar *convert(struct rectangular rec)
  {
    struct polar *polp;

    polp =  (struct polar *)malloc(sizeof(struct polar));
    if(polp)
      {
        if (rec.x == 0 && rec.y == 0)
            polp->r = polp->theta =0;
        else
          {
            polp->r = sqrt(rec.x*rec.x + rec.y*rec.y);
            polp->theta = atan2(rec.y, rec.x);
          }
      }
    return polp;
  }
```

```
struct rectangular r = {2, 1};
struct polar *pp;
if (pp = convert(r)) printf("%f %f", pp->r, pp->theta);
```

The storage allocated by calling `malloc`, unlike the storage allocated to the automatic variables, is not automatically released when the function containing the call to `malloc` exits. Hence, when `convert` returns, the storage allocated to the automatic pointer variable `polp` in `convert` is released, but the storage allocated to the structure that `polp` points to is not released. Since `convert` returns the pointer to this storage, this pointer can be used in the calling function to access the member variables of this structure.

Note that it is incorrect to write the preceding function as

```
struct polar *convert(struct rectangular rec)
  {
    struct polar pol;
    ...
    return &pol;
  }
```

WHY?