Lecture 7
Structures – Part 4

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, by Kumar

STRUCTURES CONTAINING POINTERS

A structure can contain pointers as member variables. For example, the structure definition

```
struct location
  {
    char *name;
    char *addr;
  };
```

defines a structure `location` that contains two character pointers, `name` and `addr`, as member variables.

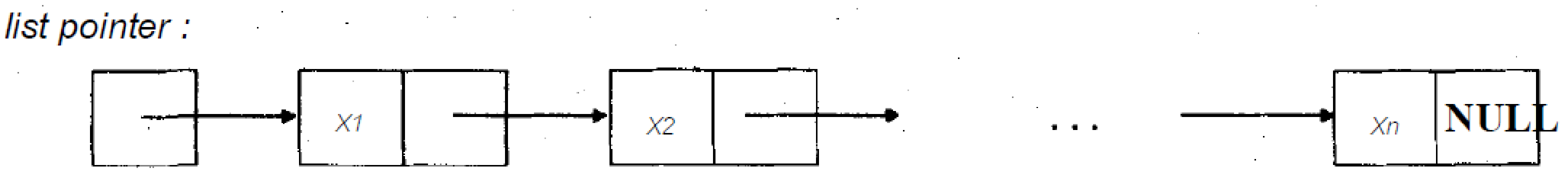
```
struct location att = '
     {"bell labs", "murray hill, new jersey"};
struct location ibm;

ibm.name = "almaden research center";
ibm.addr = "san jose, California";
printf("%s", att.name);
```

Self-Referential Structures

We mentioned in Section 8.1.6 that a structure may not be nested within itself. However, structures may contain pointers to structures of their own type. For example,

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

is a legal structure declaration, since **parent** in this example is a pointer to the **company** structure, and not an embedded occurrence of the **company** structure.



A singly linked list

Linked Lists

```
struct node
  {
    int data;
    struct node *next;
  };
```

mknode function that allocates storage for a node, initializes it, and returns a pointer to it as

```
struct node *mknode(int data)
  {
    struct node *np;
    np = (struct node *) malloc(sizeof(struct node));
    if (np)
      {
        np->data = data;
        np->next = NULL;
      }
    return np;
  }
```

A node created by calling `mknode` is not yet inserted into the linked list. The following function inserts the node into the list in such a way that all preceding nodes on the list have larger data values:

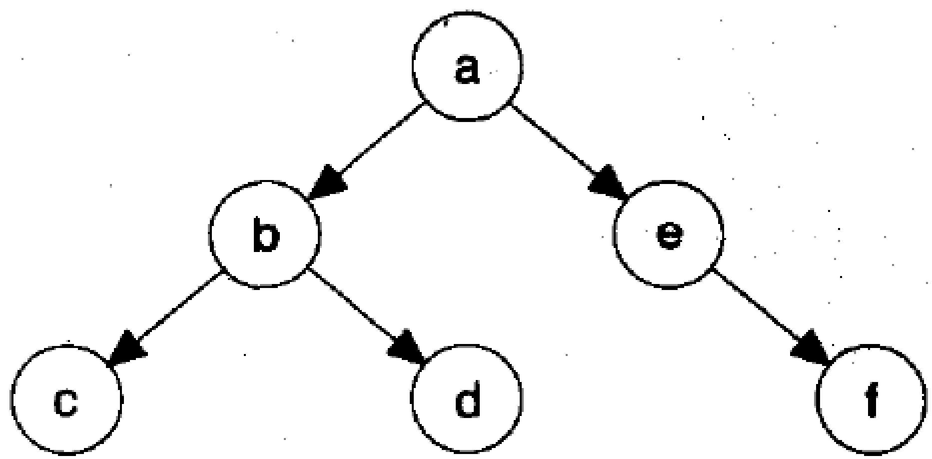
```c
struct node *insert(struct node **list, int data)
  {
    struct node *np;

    if (np = mknode(data))
      {
          struct node *curr = *list, *prev = NULL;
          /* locate the position of this node in the list */
          while(curr!=NULL && data<curr->data)
          {     prev = curr;
                curr = curr->next;
          }
         /* let this node point to the node next in the list */
         np->next = curr;

         / * let the previous node in the list point to this node * /
         if (prev)
             prev->next = np;
         else
             *list = np;  /* this node is the first in the list */
      }
    return np;
  }
```

The following function prints the data values of all the elements in the list:

```
void print(struct node *list)
  {
      for ( ; list; list = list->next)
          printf("%d ", list->data);
      printf ("\n");
  }
```

```
void sort(void)
   {
      struct node *list = NULL;
      int i;

      while (scanf("%d", &i) != EOF &&
               insert (Slist, i) )/* build list */
                     ;

      print(list);
   }
```

Trees:



A binary tree

```
struct node
  {
    char data;
    struct node *lchild;
    struct node *rchild;
  };
```

If T consists of a single node, then that node by itself is the preorder, inorder, and postorder traversal of T; otherwise

i. the preorder traversal of T is the root n of T, followed by the preorder traversal of nodes in the left subtree of n, and then the preorder traversal of nodes in the right subtree of n;

ii. the inorder traversal of T is the inorder traversal of nodes in the left subtree of root n of T, followed by the root n, and then the inorder traversal of nodes in the right subtree of n, and

iii. the postorder traversal of T is the postorder traversal of nodes in the left subtree of root n of T, followed by the postorder traversal of nodes in the right subtree of n, and then the root n.

UNIONS

The *union* is a construct that allows different types of data items to share the same block of memory. The compiler automatically allocates sufficient space to hold the largest data item in the union. However, it is the programmer's responsibility to keep track of what is currently stored in the union.

The syntax for defining and accessing a union is similar to that for structures, except that the keyword `union` is used in place of `struct`. For example, the statement

```
union chameleon
  {
    double d;
    int i;
    char *cp;
  } data;
```

defines a variable `data` that can hold either a `double`, an `int`, or a pointer to `char`.

When `data` needs to be accessed as a `double`, it is accessed as

```
data.d
```

when it needs to be accessed as an `int`, it is accessed as

```
data.i
```

and when it needs to be accessed as a pointer to `char`, it is accessed as

```
data.cp
```

Although a union contains sufficient storage for the largest type, it may contain only one value at a time; it is incorrect to store something as one type and then extract as another. Thus, the following statements

```
data.d = 1.0;
printf("%s", data.cp);
```

produce anomalous results.

To keep track of what is currently stored in `data`, another variable `dtype` may be defined. Whenever a value is assigned to `data`, the variable `dtype` is also set to indicate its type. Later in the program, `dtype` may be tested to determine the type of the value in `data`. Thus, following the definitions

```
♦define DOUBLE      1
♦define INT         2
♦define CP          3
int dtype;
```

the statements

```
data.cp = "anolis";
dtype = CP;
```

```
switch (dtype)
  {
    case DOUBLE:
        printf("%f", data.d);
        break;
    case INT:
        printf ("%d", dat.a.i);
        break;
    case CP:
        printf("%s", data.cp);
        break;
    default:
        printf ("unknown type of data");
  }
```

```
struct item
  {
    int itemno;
    struct
      {
        int stype;
        union
          {
            struct
              {
                char *streetaddr;
                char *city;
                char *state;
              } domestic;
            struct
              {
                char *country;
                char *completeaddr;
              } foreign;
          } addr;
        float price;
      } suppliers[MAXSUPPLIERS];
  } items[MAXITEMS];
```

The method of accessing a member of a union in a structure or that of a structure in a union is identical to the method of accessing a member of a structure in a structure. For example, the following program fragment prints the names of the countries of all the foreign suppliers:

```
for (i = 0; i < MAXITEMS; i++)
   for (s = 0; s < MAXSUPPLIERS; s++)
      if (stype == FOREIGN)
         printf ("%s\n",
               items[i].suppliers[s].addr.foreign.country);
```

As in the case of structures, unions may not contain instances of themselves, although they may contain pointers to instances of themselves.