C Programming

Recitation 7

May 2017

WHY DO WE NEED STRUCTURES?

- Enable grouping related data items of different types (e.g. inventory record of stock item) and referring them using the same name.
- Allow treating a fixed number of data items (possibly different types) as a single object.
- Create advanced data structures like linked lists, trees etc.

WHAT IS A STRUCTURE?

- **Structure**: Collection of logically related data items grouped together under a single name called a *structure tag*.
- Data items that make up a structure are called its *members*, *components* or *fields*.
- In the code, *data* is the name of the structure.

```
struct data {
/* member declarations */
};
```

SYNTAX OF STRUCTURES

```
1  struct item {
2      int itemno;
3      float price, quantity;
4  };
5
6  struct order {
7      int itemno;
8      float price, quantity;
9  };
10
11 int itemno;
12
13 struct day {
14     int day, year;
15 };
16
```

- Members of a structure can be different (lines 2-3) or same type (line 14).
- The name of a member in a structure can be same as its tag (lines 13,14).
- The name of a member in a structure can be same as some non-member variable (lines 7,11).
- 2 member variables in different structures can have same name (lines 3,8).

STRUCTURE VARIABLES

```
1  struct date {
2        int day, month, year;
3  } order_data, arrival_data;
4
5  struct date val;
6
7  struct {
8        float salary;
9        int ssn, age;
10 } emp1, emp2; /* OK */
11
12 struct emp3; /* Wrong */
13
```

- Variable(s) can be declared immediately after definition (line 3).
- Variable(s) can be declared in separate line (line 5).
- It is legal to define a structure without a tag (line 7). In this case, all the variables should be declared immediately after definition (line 10); it is illegal to declare a variable in separate line (line 12).

STRUCTURE VARIABLES

```
struct {
      int a ; float b;
} p;

struct {
      int a ; float b;
} q;

struct x {
      int a ; float b;
} r;

struct y {
      int a ; float b;
} s;

struct y t;
```

- Although the members are of the same name and type, p, q, r and s are of different types.
- s and t are of the same type.

INITIALIZATION

```
1  struct date {
2          int day;
3          int month;
4          int year;
5  } turing = {23,06,1912};
6
7  struct date neumann = ↩
       {28,12,1903};
8
9  /* OK */
10 struct date x = {13,2};
11
12 /* Wrong */
13 struct date y = {1,1,2017,5};
14
15
```

- A struct variable can be initialized either immediately after definition (line 5) or in separate line (line 7).
- If there are fewer initializers than member variables, the remaining member variables are initialized to 0. In line 10, *year* is initialized to 0.
- It is illegal to provide more initializers than member variables (line 13).

ASSIGNMENT AND ACCESSING MEMBERS

- A structure variable may be **assigned** to another structure variable of the same type
- "." operator is used to **access** the member of a structure.

```
1  struct date {
2        int day, month, year;
3  } man_on_moon = {20,7,1969};
4
5  /* Assignment */
6  struct date american, revolution = {4,7,1776};
7  american = revolution;
8
9  /* Accessing members */
10 printf("%d\n", man_on_moon.year) ;
11 printf("%d\n", american.year) ;
```

CHECKPOINT

- Let's download *ex1.c* and try to complete the tasks.

```c
#include <stdio.h>

/* define a structure type struct birthday */

/* declare friendBday to be a variable of this type */

int main(void)
{

/* declare myBday to be of type struct birthday
and initialize it */

/* assign myBday to friendBday */

/* access members of friendBday and print */

return 0;
}
```

NESTED STRUCTURES

```
struct animal {
    int age;
};

struct pet {
    struct animal dog;
};

struct person {
    struct pet myPet;
};

struct person murat = {{{1}}};

printf("%d\n", murat.myPet.dog.age);
```

- Structures can be nested and there is no limit on depth of nesting.
- However, a structure can not be nested within itself.
- A particular member inside a nested structure can be accessed by repeatedly applying the dot operator.

POINTERS TO STRUCTURES

```
struct date {
      int day, month, year;
} bDay = {1,1,1990};

struct date *newDay = &bDay;

printf("%d\n", (*newDay).year);

newDay->year=1994;
printf("%d\n", newDay->year);
```

- A pointer to a structure is created in the same way such as int or char is created.
- There are 2 ways to access members of a structure pointed by a pointer: Either using dereferencing operator or the structure pointer(or arrow operator) "->".

ARRAYS & STRUCTURES

- A structure can have an array as its member.
- One can create an array of a structure.
- It is possible to mix the two.

```
struct date {
        int day, month, year;
} birthdays[2] = { {23, 6, 1912}, {28, 12, 1903} };

struct ndate {
        int day, year;
        char monthname[5];
} bDays[2]= { {23, 1912, {"Jun"} }, {28, 1903, {"Dec"} } };

printf("%d\n", birthdays[0].day);
printf("%s\n", bDays[1].monthname);
```

FUNCTIONS & STRUCTURES

- There are 2 ways to use structures as arguments: Either using structure directly or a pointer to the structure.
- In the first case, the whole struct is copied and is slow. (pass by value semantics)
- In the second case, the address of the struct is copied so it works much faster. (pass by reference semantics)

```
struct date {
      int day, month, year;
} wday = {1, 5, 2017} ;
int check (struct date day, int yearVal)
{
      return day.year==yearVal ? 1 : 0 ;
}
int checkP (struct date *day, int yearVal)
{
      return day->year==yearVal ? 1 : 0 ;
}
```

```
check (wday, 2015);
checkP (&wday, 2017);
```

Checkpoint

- Let's download *ex*2.*c* and examine it.

STRUCTURES CONTAINING POINTERS

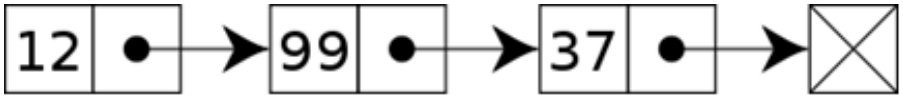
- A structure can contain pointers as member variables.
- A structure can contain pointers to structures of its own type.
- Structures which contain pointers to structures of their own type provide a basis for several useful data structures such as linked list, tree, stack etc. trees.

```
struct location
{
        char *name, *addr;
} att = {"bell labs", "new jersey"};

struct node
{
        int val;
        struct node *next; /* legal */
};

printf("%s", att.name);
```

LINKED LISTS

- A **singly linked list** is made up of nodes, each node consisting of an element of the list and a pointer to the next node on the list.
- The first element and the last element of the list is called the **head** and **tail**, respectively.
- The advantage of linked lists is that the list elements can be inserted and deleted by adjusting the pointers without shifting other elements, and storage for a new element can be allocated dynamically and freed when an element is deleted.

LINKED LISTS

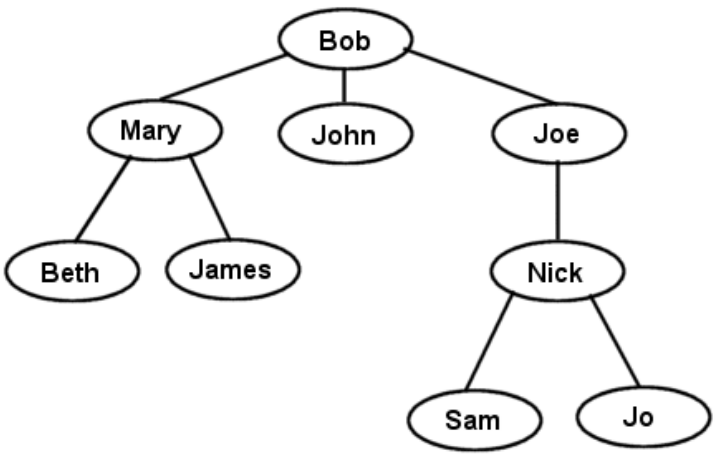


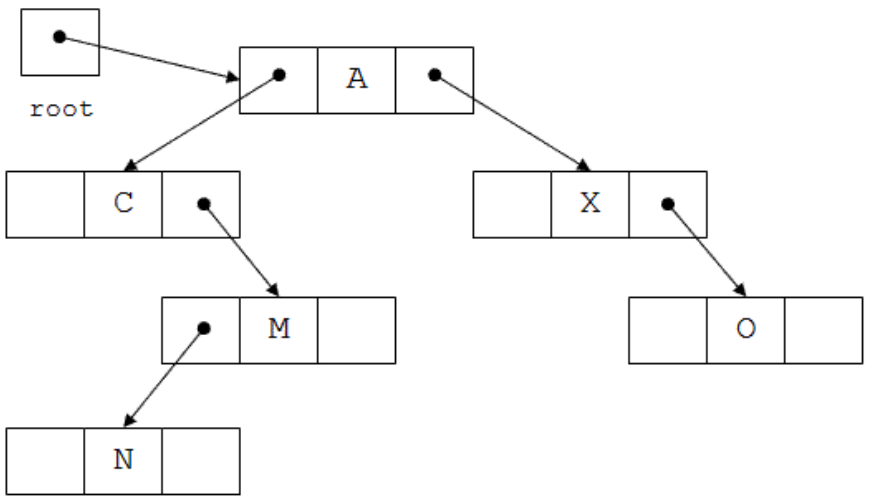

```
struct node
{
    int value;
    struct node *next;
};

struct node *make_node(int value)
{
    struct node *tmp = (struct node*)(malloc(sizeof(struct node)));
    tmp->next = NULL;
    tmp->value = value;
    return tmp;
}

struct node *start = make_node(12);
start->next = make_node(99);
start->next->next = make_node(37);
```

TREES

- A **tree** is defined recursively as a collection of nodes (0 or more) where each node except root node is partitioned into disjoint subsets called subtrees.
  - None of the nodes are pointed more than 1 node.
  - Root node is not pointed by any node.
- A **binary tree** is a special kind of tree in which each node has at most 2 children, namely left and right child.

TREES





CHECKPOINT

- Let's download *ex*3.*c*, *ex*4.*c* and examine them.

BASICS OF UNION

- **Union:** is a construct that allows different types of data items to share the same memory location.
- Compiler automatically allocates sufficient space to hold the largest data item in the memory.
- It contains only one value at a time.
- Members of a union is accessed by (.) operator.

```
#include <stdio.h>
union item
{
        int a;
        float b;
};

int main( )
{
        union item it;
        it.a = 12;
        it.b = 20.2;
        printf("%d\n",it.a);
        printf("%f\n",it.b);
        return 0;
}
```