Structures: Initialization, Passing to functions, Self-referential structures, Unions

The syntax for initializing structures is similar to that for initializing arrays. A structure variable in a declaration can be initialized by following it with an equal sign and a list of constants contained within braces. If not enough values are used to assign all the members of the structure, the remaining members are assigned the value zero by default (NULL is assigned for a pointer). In the code below, the alias `student` can be thought of just like `int` or `double`. The declarations `int p` or `int q[3]` define `p` to be an integer and `q` to be an array of three integers. Similarly, `student sp` and `student list[3]` declare `sp` and `list` to be of the type `student` (a structure) and an array of the type `student` of dimension 3 (an array of structures). These are initialized as shown below. Array element `list[i]` ($0 \leq i \leq 2$) is like a `student` variable.

```
#include<stdio.h>
typedef struct {
int dd,mm,yy;
} date;
typedef struct {
char name[40];
int roll;
date dob;
} student;
int main()
{
 int i;
 student sp={"S Pal",26521,19,12,2010};
 student list[3]={{"A Roy",26522,18,9,2010},{"B Das",26422,8,2,2009},
 {"A Shah",25521,10,7,2011}};
 printf("Name   : %s\n",sp.name);
 printf("Roll no: %d\n",sp.roll);
 printf("DoB    : %02d/%02d/%d\n",sp.dob.dd,sp.dob.mm,sp.dob.yy);
 printf(".................\n");
 printf("List of students:\n");
 printf(".................\n");
 for(i=0;i<3;i++)
 {
 printf("Name   : %s\n",list[i].name);
 printf("Roll no: %d\n",list[i].roll);
 printf("DoB    : %02d/%02d/%d\n",list[i].dob.dd,list[i].dob.mm,list[i].dob.yy);
```

```
 }
 return 0;
}
```

Listing 66: C Program "inistrc.c"

Below is the output:

```
Name   : S Pal
Roll no: 26521
DoB    : 19/12/2010
.................
List of students:
.................
Name   : A Roy
Roll no: 26522
DoB    : 18/09/2010
Name   : B Das
Roll no: 26422
DoB    : 08/02/2009
Name   : A Shah
Roll no: 25521
DoB    : 10/07/2011
```

If we write `student sp={0};` then it initializes all members of the structure to have value zero. Similarly, `student list[3]={0};` causes all members of all element of the `list` to have zero values.

In C, a structure is stored as a contiguous block of memory in which each member follows the next in the order they are declared. However, the actual size is often larger than the sum of its parts due to how the compiler handles hardware efficiency.

Passing structures to functions: We have already written a program in which we pass a structure as argument (see Listing 64 of Lecture 23) and the function returns a structure of the same type. When we pass a structure as an argument of a function, the components of the structure are copied to the corresponding local structure in the function. Similarly, the components of the structure are copied to the structure of the calling function when the the function returns a structure. An alternate option is to pass the address of the structure variable. We know that the members of a structure can be accessed using a pointer variable which points to the structure. The advantage is that we send only the address of the structure instead of its components. We write the same program as given in Listing 64 of Lecture 23 but the arguments of the function is now pointers.

```
#include<stdio.h>
#include<stdlib.h>
typedef struct {
int dd,mm,yy;
} date;
void date_of_tomm(date *,date *);
int main()
{
 date today, tomorrow;
 printf("Enter three integers representing dd mm yyyy of today:");
 scanf("%d%d%d",&today.dd,&today.mm,&today.yy);
 date_of_tomm(&today,&tomorrow);
 printf("Date of next day : %02d/%02d/%d\n",tomorrow.dd,tomorrow.mm,tomorrow.yy);
 return 0;
}
void date_of_tomm(date *p, date *q)
{
 if((p->dd < 1) || (p->dd > 30) || (p->mm != 4) || (p->yy != 2026))
 {
  printf("Invalid date\n");
  exit(1);
 }
 q->yy=p->yy;
 if(p->dd==30)
 {
  q->dd=1;
  q->mm=p->mm+1;
 }
 else
 {
  q->dd=p->dd+1;
  q->mm=p->mm;
 }
 return;
}
```

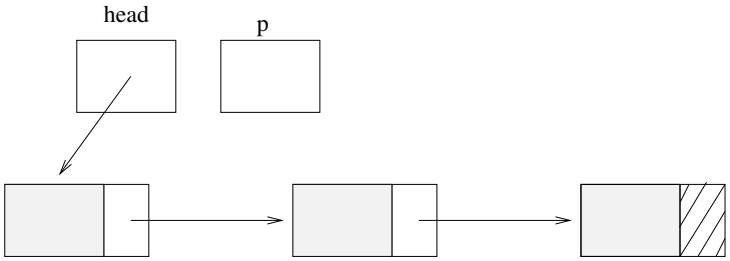
Listing 67: C Program "pdatnxt.c"

We get the same input-output as that of Listing 64 of Lecture 23. Note that we have passed the addresses of `today` and `tomorrow`. The local variables `p` and `q` are assigned these addresses.

Hence, `p->dd` is equivalent to `today.dd` and so on. Thus, at the conclusion of the function call, `tomorrow` contains the date of the next day.

The above function is of type `void`, which takes two addresses of type `date`. In Listing 64 of Lecture 23, we wrote the function that takes an argument of the type `date` and returns a `date`. Obviously, we can take other combinations as well. For example, the argument can be an address of type `date` and returns a `date` or a void function with one argument of type `date` and the address of `date` as the other argument.

Self-referential structure: A self-referential structure is a data structure that contains one or more pointers pointing to another instance of the same structure type. Instead of holding a copy of itself (which is not allowed as it would create an infinite size), it holds a reference (memory address) to a similar object, allowing the creation of dynamic, interconnected "chains" or "trees" of data. Consider the program in Listing 66, in which we created an array `list` of three elements of type `student`. The same can be implemented using a self-referential structure. The schematic diagram shown here implements a list of three structures.



The unshaded portions of the structure contain the addresses of the same structure type. The shaded portions represent the other components of the structure. The unshaded portion of the last structure contains `NULL`, meaning that it is the end of the list. We also have a variable `head` which contains the address of the first structure. Note that we do not change `head` since it contains the address of the first element of the list. Also, the variable `p` can also hold the address of a structure. This variable `p` is used to traverse along the list. We create a list of three students in the following program.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct {
int dd,mm,yy;
} date;
typedef struct st {   // tag needed for self-referential structure
char name[40];
int roll;
date dob;
struct st *next;
```

```
} student;
int main()
{
 student *head,*p;    //head & p are pointers of the type student

 head=(student *)calloc(1,sizeof(student)); // create space for 1st struct and
                                            // assign its address to head
 p=head; // p has the same address as head & we work with p from now
        // use p to assign components of 1st struc
 strcpy(p->name,"A Roy");
 p->roll=26522;
 p->dob.dd=18;
 p->dob.mm=9;
 p->dob.yy=2010;
 p->next=(student *)calloc(1,sizeof(student)); // create space for 2nd struct and
                                               // assign its address to 1st struc

 p=p->next;                      // p has the address of 2nd struc
                           // use p to assign components of 2nd struc
 strcpy(p->name,"B Das");
 p->roll=26422;
 p->dob.dd=8;
 p->dob.mm=2;
 p->dob.yy=2009;

 p->next=(student *)calloc(1,sizeof(student)); // create space for 3rd struct and
                                          // assign its address to 2nd struc

 p=p->next;                      // p has the address of  3rd struc
          // use p to assign components of 3rd struc
 strcpy(p->name,"A Shah");
 p->roll=25521;
 p->dob.dd=10;
 p->dob.mm=7;
 p->dob.yy=2011;
 p->next=NULL;      // end of the list
   // print the list of students
 printf(".................\n");
 printf("List of students:\n");
```

```
 printf("................\n");

 p=head;   // start with address of 1st struct
 while(p != NULL)        // iterate until end of list encountered
 {
        printf("Name   : %s\n",p->name);
        printf("Roll no: %d\n",p->roll);
        printf("DoB    : %02d/%02d/%d\n",p->dob.dd,p->dob.mm,p->dob.yy);
        p = p->next;    // move to next struct
 }
 return 0;
}
```

Listing 67: C Program "simplist.c"

Here is the output of the program as expected.

```
................
List of students:
................
Name   : A Roy
Roll no: 26522
DoB    : 18/09/2010
Name   : B Das
Roll no: 26422
DoB    : 08/02/2009
Name   : A Shah
Roll no: 25521
DoB    : 10/07/2011
```

We need to use a tag `st` in the structure for the `student` to define the self-referential structure. The number of structures in the list is not specified in advance. Since the structures are created using `calloc()` (or `malloc()`), these spaces are permanent during the run-time of the program. Thus, they can be created inside a function, too. On the other hand, to free the spaces allocated for these structures, we need to free individual spaces for each structure.

Unions: A `union` in C is a user-defined data type that allows us to store different data types in the same memory location. Unlike a `struct`, which allocates separate memory for each member, a `union` only allocates enough space for its largest member, and all members share that space. Since all members share the same memory, changing the value of one member overwrites the value of the others. The size of a union is primarily determined by its largest member, but it often includes trailing padding to satisfy memory alignment requirements. Union members

can be accessed using the dot (.) operator similar to `struct`. Unions can be defined similar to structures. Unions are useful to save memory by storing different types of data in the same memory space. To explain this consider the code given below.

```
#include<stdio.h>
typedef union {
char c;
int x;
double y;
} data;
int main()
{
 data P;
 printf("Size of union data is %lu bytes\n",sizeof(P));

 P.c='A';
 printf("Char component is %c\n",P.c);
 P.x=200;
 printf("Char component is %c\n",P.c);
 printf("Int component is %d\n",P.x);
 P.y=427.5;
 printf("Char component is %c\n",P.c);
 printf("Int component is %d\n",P.x);
 printf("Real component is %0.2f\n",P.y);
 return 0;
}
```

Listing 68: C Program "uni.c"

Below is the output of the code.

```
Size of union data is 8 bytes
Char component is A
Char component is ?
Int component is 200
Char component is
Int component is 0
Real component is 427.50
```

Out of the three members, real component `P.y` takes the highest size of `8` bytes. Hence, `8` bytes is the size of the union `P`, which is confirmed by the first output. Note that we first assign

character `A` to `P.c`. Now when we assign integer `200` to `P.x`, then `P.c` becomes undefined. This is evident from the output. Thus, whenever we assign a value to a certain member of the `union`, the other members have undefined values.

Return to Lecture Topics