Structures: Introduction, Accessing members of structures

We have seen that there are three fundamental data types consisting of integer (e.g., `int`, `long int`, etc.), real (e.g., `float`, `double`, etc), and character (`char`). In C, it is possible to define data types from the fundamental data types. We have already encountered one such derived type: an array that is used to represent homogeneous data. In contrast, the structure type discussed here is used to represent heterogeneous data. A structure has components, called members, that are individually named. Since the members of a structure can be of different types, the programmer can create data that is suitable for a specific application.

C typedef: In C, the `typedef` keyword creates an alias (a new name) for an existing data type. It does not create a new data type itself; it simply provides a shorthand or a more descriptive name for one that already exists. Here is an example of a `typedef` in a C program. Usually, the `typedef` is declared above the `main()` so that all the functions can have access to it. The following program is self-explanatory.

```
#include<stdio.h>
#include<math.h>
#define M 10
#define N 20
typedef int integer; // integer for int
typedef double dbl;  // dbl for double
typedef dbl vector[N]; // vector for 1D array of dim N
typedef vector matrix[M]; // matrix for 2D array of dim MxN
void multip(matrix,vector,vector,int,int,int,int);
int main()
{
 int i,p,q;
 integer j,n; // can define int using integer too
 vector u,v; // u & v are of dim N
 matrix A;   // A is of dim MxN
 printf("Enter actual dimension of vector :");
 scanf("%d",&n);
 printf("Enter components of vector v :");
for(i=0;i<n;i++)
 {
  scanf("%lf",&v[i]);
 }

 printf("Vector v:\n");
```

```
 for(i=0;i<n;i++)
 {
  printf("%10.4lf",v[i]);
 }
 printf("\n");
 printf("Enter actual row & col dims of matrix A:");
 scanf("%d%d",&p,&q);
 printf("Enter matrix row wise:\n");
 for(i=0;i<p;i++)
 {
  for(j=0;j<q;j++)
  {
   scanf("%lf",&A[i][j]);
  }
 }
 printf("Matrix A is:\n");
 for(i=0;i<p;i++)
 {
  for(j=0;j<q;j++)
  {
   printf("%10.4f   ",A[i][j]);
  }
  printf("\n");
 }
 multip(A,v,u,n,p,q,flag);
 return 0;
}
```

Listing 62: C Program "atyped.c"

The above program is self-explanatory. It reads a matrix `A` and a vector `v`. If matrix-vector multiplication is not possible, then it prints an appropriate message. Otherwise, it prints the vector `u`, where `u=Av`. Below is a typical input-output of the code.

```
Enter actual dimension of vector :4
Enter components of vector v :1 2 3 4
Vector v:
    1.0000    2.0000    3.0000    4.0000
Enter actual row & col dims of matrix A:2 4
Enter matrix row wise:
```

```
4 3 2 1
1 2 3 4
Matrix A is:
    4.0000       3.0000       2.0000       1.0000
    1.0000       2.0000       3.0000       4.0000
Vector u:
   20.0000   30.0000
```

The declaration `typedef dbl vector[N]` is equivalent to `typedef double vector[N]`. Similarly, `typedef vector matrix[M]` is equivalent to `typedef double matrix[M][N]`. We can either use the fundamental type `int` or its alias `integer`. Similarly, for `dbl` and `double`. The declaration `matrix A` is equivalent to `double A[M][N]`. However, we rarely use `typedef` for `int`, `double`, etc., as shown in Listing 62. On the other hand, `typedef` is widely used for `structures` as we shall shortly see.

Structures: Let us construct a structure that describes a date. A date can be represented by three integers: day, month, and year. We declare the structure type

```
struct date {
int dd;
int mm;
int yy;
};
```

Note the semicolon at the end of the closing curly bracket. Instead of writing in three separate lines, we could have also written

```
struct date {
int dd,mm,yy;
};
```

In this declaration, `struct` is a keyword, `date` is the structure tag name, and the variables `dd, mm` and `yy` are members of a structure. This declaration creates the derived data type `struct date`. It is an example of a user-defined data type. This declaration can be thought of as a template–it creates the `struct date`, but no storage is allocated. The tag name `date`, along with the keyword `struct`, can now be used to declare variables of this type. For example,

```
struct date today, tomorrow;
```

This declaration allocates storage for the identifiers `today` and `tomorrow`, which are of type `struct date`. An alternative scheme is to write

```
struct date {
int dd,mm,yy;
} today, tomorrow;
```

which defines the type `struct date` and declares `today` and `tomorrow` to be of this type, all at the same time.

To access the members of a structure, we use the member access operator "·". To assign the date 15-4-2026 to the identifier `today`, we write

```
today.dd=15;
today.mm=4;
today.yy=2026;
```

Now, let us write a simple code that reads a date of April 2026 and writes the date for the next day. The structure has been defined above the `main()` so that all the functions of the program know about the structure. Once we declare the template `struct date`, we can think of `struct date` just like `int` or `double`. For example, `int i` declares `i` to be of integer type. Similarly, `struct date w` declares `w` to be a structure which has three members of `int` type. In the `main()`, we define `today` and `tomorrow` of the type `struct date`. Note that `today.dd` is just like an integer. To read this member, we use `%d` in `scanf()`. Since "·" has higher priority than "&", the expression `&today.dd` is equivalent to `&(today.dd)`. Next, we write a function `date_of_tomm()` which takes an argument of the type `struct date` and returns a variable of the type `struct date`. The structure variables `p` and `q` are local to the function. The variable `p` receives a copy of the structure `today`. At the end of the function, `q` contains the date of the next day, which is returned and assigned to the variable `tomorrow`.

```
#include<stdio.h>
#include<stdlib.h>
struct date {
 int dd,mm,yy;
};
struct date date_of_tomm(struct date);
int main()
{
 struct date today,tomorrow;
 printf("Enter three integers representing dd mm yyyy of today:");
 scanf("%d%d%d",&today.dd,&today.mm,&today.yy);
 tomorrow=date_of_tomm(today);
 printf("Date of next day : %02d/%02d/%d\n",tomorrow.dd,tomorrow.mm,tomorrow.yy);
 return 0;
```

```
}
struct date date_of_tomm(struct date p)
{
 struct 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 q;
}
```

Listing 63: C Program "simpst.c"

Below is some sample input-output:

```
Enter three integers representing dd mm yyyy of today:12 4 2026
Date of next day : 13/04/2026

Enter three integers representing dd mm yyyy of today:30 4 2026
Date of next day : 01/05/2026

Enter three integers representing dd mm yyyy of today:2 4 2027
Invalid date
```

Instead of writing `struct date` to declare a variable representing the structure, we can use a short term using `typedef`. For this purpose, see the following code segment.

```
struct date {
int dd,mm,yy;
};
typedef struct date date;
```

By the above `typedef`, now `date` is equivalent to `struct date`. Note that we have used tag name as the `typedef` name, but it is allowed. You can also use a different name, like `typedef struct date DATE`, as well, for which `DATE` and `struct date` are equivalent. The following without a tag is more compact, which we shall use from here onward.

```
typedef struct {
int dd,mm,yy;
} date;
```

Here, we do not use tag but `date` is equivalent to `struct date` defined earlier. Below is the same code given in Listing 63 using the new convention. Here, we also use a pointer variable `s` which can hold the address of a variable of the type `date`. This pointer variable is used to print the date of the next day. Note that `s=&tomorrow` assigns the address of `tomorrow` to `s`. Hence, `*s` represents the structure `tomorrow` and `(*s).dd` represents the component `dd` of `tomorrow`. Note that the parentheses is necessary since dot "." operator has higher precedence than the de-referencing "*" operator. In absence of parentheses, `*s.dd` is equivalent to `*(s.dd)` which has no meaning. The expression `(*s).dd` can also be written as `s->dd`, where "`->`" is called indirect structure member selection operator. This is widely used and we follow this while accessing members of a structure using pointer to that structure.

```
#include<stdio.h>
#include<stdlib.h>
typedef struct {
int dd,mm,yy;
} date;
date date_of_tomm(date);
int main()
{
 date today, tomorrow,*s;       // s is a pointer to structure
 s = &tomorrow;         // s points to tomorrow
 printf("Enter three integers representing dd mm yyyy of today:");
 scanf("%d%d%d",&today.dd,&today.mm,&today.yy);
 tomorrow=date_of_tomm(today);
 // print date of next day using pointer s
 printf("Date of next day : %02d/%02d/%d\n",s->dd,s->mm,s->yy);
```

```
 return 0;
}

date 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 q;
}
```

Listing 64: C Program "asimpst.c"

We can have structure(s) within another structure. Consider a student who can be identified by name, roll number and date of birth. Note that a name can be stored in a string, a roll number is an integer, and the date of birth is a variable of the type `date` discussed above. The program below implements a code that assigns members of the student and then prints them. The specific student is S Pal with roll no. 26521 and date of birth 19-12-2010. We first define a variable of type `date` that represents a date structure. Then we define a variable of the type `student` which represents the structure corresponding to a student. In the definition for `student`, we have used the variable `dob`, which is of the type `date`. In the `main()`, the variable `sp` is of the type student. Then `sp.dob` represents a date and `sp.dob.dd` represents the day of the date of birth and so on. Since `.` operator associates from left to right, there is no need for parentheses.

```
#include<stdio.h>
#include<string.h>
typedef struct {
int dd,mm,yy;
} date;
typedef struct {
char name[40];
int roll;
date dob;
} student;

int main()
{
 student sp;
 strcpy(sp.name,"S Pal");
 sp.roll=26521;
 sp.dob.dd=19;        // assign dd of dob of sp
 sp.dob.mm=12;        // assign mm of dob of sp
 sp.dob.yy=2010;      // assign yy of dob of sp
 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);
 return 0;
}
```

Listing 65: C Program "stdnt.c"

Below is the output.

```
Name   : S Pal
Roll no: 26521
DoB    : 19/12/2010
```

Finally, we extend the operator precedence and associativity table discussed in Lecture 17 with the new operators (·, `->`) introduced here. Note that ·, `->`, introduced here, have equal and highest precedence.

Category	Operator	Associativity
Postfix operators	`( )` (parenthesis), `[ ]` (array subscript), `->` (indirect structure member selection), `.` (structure member selection), `++` (postfix increment), `--` (postfix decrement),	left to right
Unary operators	`+` (unary plus), `-` (unary minus), `++` (prefix increment), `--` (prefix decrement), sizeof, (type) (type casting), `!` (logical not), `*` (de-reference), `&` (address-of)	right to left
Multiplicative operators	`*` (multiplication), `/` (division), `%` (modulus)	left to right
Additive operators	`+` (addition), `-` (subtraction)	left to right
Relational operators	`<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to)	left to right
Equality operators	`==` (equal to), `!=` (not equal to)	left to right
Logical and	`&&`	left to right
Logical or	`||`	left to right
Conditional (ternary) operator	`?:`	right to left
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right to left

Return to Lecture Topics