Lesson 9 STRUCTURES IN C

Introduction to Structures

A structure is a user defined data type. It is a combination of several different previously defined data types, including other structures. A simple definition of a structure is as follows,

"A structure is a collection of one or more variables, possibly of different type, grouped together under a single name"

Consider a point in the plane that you may come across in geometry. To plot the point we need to know its x-axis coordinates and its y-axis coordinates. Thus the point consists of two values (x,y). We can define a structure called "point" as follows

```
struct point{
             int x;
             int y;
            };
```

Defining a Structure

The code above is a declaration of the structure point. We must declare a structure before we use it. The declaration tells the compiler what the structure looks before variables of that type are used. A structure declaration must contain the keyword **struct**, the name or tag of the structure, the contents of the structure enclosed in brackets, followed by a semi colon.

Declaring a structure variable

The line of code below declares two variables M and N of type point (structure point).

```
struct point M,N;
```

Space is now set aside in memory to store the two variables.

Accessing Components in a Structure

To assign a value to the x coordinate and y coordinate, we must access the structure. The following is how we achieve this

```
 M.x   = 2;
 N.y   = 9;
```

We have used the dot operator to carry out this task. Consider the following structure below. The structure name is called data and contains the name of an employee in the company together with their weekly salary in pounds. Two variables, employee1 and employee2 are declared to be of type data

```
struct data {
             char name[50];
             int salary;
             } employee1,employee2;
```

To enter information into employee1 and employee2 from the console, we can use the following

```
scanf("%s",&employee1.name);
scanf("%d",&employee1.salary);
scanf("%s",&employee2.name);
scanf("%d",&employee2.salary);
```

The structure defined above could be enlarged to include information such as date of birth, pps number, address, grade, tax band, role etc.

Initialising a Structure

As with arrays, structures can be initialised. For example, in the structure *point* we can initialise the point M = (0,0) by writing *struct point M={0,0};* Similarly for the structure *data* we can initialise employee1 by writing *struct data employee1 = {"Wilfred Owen", 350};*. More complicated structures are initialised in a similar fashion.

Using Assignment Statements with Structures

The value of one structure variable may be assigned to the value of another structure variable of the same type. For example ANSI C allows the assignment statement

```

employee1 = employee2;
```

All the values in employee2 are now assigned to employee1, they are equal.

Nesting Structures

The structures we have seen so far have been very simple. It is possible to define structures containing hundreds of elements. However, to define all of the elements in one structure may be a little confusing. To avoid this we could use a hierarchical structure definition. Suppose that we want to define a new structure say of type *box* which contain the four coordinates of the box each of type *point*. We can use the previously defined structure point nested inside our new structure box. The code below is an example of this. Note that the definition of the structure point but be visible to the new structure box.

```
struct box {
            struct point M;
            struct point N;
            struct point Q;
            struct point S;
            };
```

We can now declare a variable of type box.

```
struct box firstbox;
```

To access the x any y coordinates of each corner of the box we can use the following statements:

```
firstbox.M.x = 1;
firstbox.M.y = 2;
firstbox.N.x = 6;
firstbox.N.y = 6;
firstbox.Q.x = 1;
firstbox.Q.y = 2;
firstbox.S.x = 6;
firstbox.S.y = 6;
```

Passing Structures to a Functions

When discussing structures and functions in C, we will concentrate on three different approaches

1. passing an entire structure to a function
2. passing components of a structure to a function
3. passing a pointer to a structure to a function

1. Passing an entire structure to a function

Suppose we want to write a function called definepoint. The function enters a pair of integers into a point and returns a structure. We can use the following code for this purpose

```
struct point definepoint(struct point M,int x1,int y1)
                  {
                 M.x = x1;
                 M.y = y1;
                 return(M);
                }
```

The function accepts the structure of type point and the integers *x1* and *y1*. In the function definepoint the integers are assigned to the structure, namely the values x and y. The function then returns a structure to the main program with the newly assigned data.

Passing components of a structure to a function

The above code can be altered so that only two integers are passed down to the function definepoint and the fuction returns a structure. You should write out the code as an exercise.

Passing a pointer to a structure down to a function

In C, if you wish to pass a large structure down to a function, it is often more economical to pass a pointer to the structure instead (why?). A pointer to a structure is defined in a similar way as pointers to other types.

```
struct point *p;
```

The code above defines a pointer to a structure of type point. The assignment statement

```
p=&M;
```

Assigns the address of the point M to the pointer p. To reference the elements of the structure using pointers, the following notation is used:

```
 p->x = 4;
```

An alternative notation can also be used:

```
 (*p).x =4;
```

However, the former is by far the more popular.

So dealing with C structures and accessing their members, it is worth remembering the following.

1. the **dot operator(.)** connects a structure with a member of the structure
2. the **-> operator** connects a pointer with a member of the structure

Arrays of Structures

We have already defined the structure called data previously. Consider the personnel department in a large tech company. Suppose they would like to hold details of all their employees on record. To do this, an array of structures of type data is a possible solution. This can be easily done using the following code. An array of type data (structure) of size 100 is constructed.

```
struct data personnel[100];
```

Each element of the array is a structure of type data with. We can use the dot operator and array index to assigned values to each element as follows:

```
personnel[i].name = " Tom";
personnel[i].salary = 295;
```

This is interpreted as follows, the i refers to the array index which is a the i`th structure of type data. The dot operator is then used to access the elements of the i`th structure. Type in and compile the following code for an example of an array of structures. You should make sure that you fully understand how the program works.

```
#include<stdio.h>
struct child
 {
  char initial;
  int  age;
  int  grade;
 } children[12], *point, extra;

/* a structure called child is defined and 3 variables of this type are
   declared - an array size 12 called children, a variable called extra and
   a pointer variable called point
*/

int main()
{
 int index;
 for(index = 0 ; index < 12 ; index++)
    {
     point = children + index;
     point->initial = 'A' + index;
     point->age = 16;
     point->grade = 84;
    }

  children[3].age = children[5].age = 19;
  children[2].grade = children[6].grade = 90;
  children[4].grade = 67;

  for(index = 0 ; index < 12 ; index++)
    {
     point = children + index;
     printf("%c is %d years old and got a grade of

                %d\n",(*point).initial,children[index].age,point->grade);
    }
  extra = children[2];                 /* Structure assignment */
  extra = *point;                      /* Structure assignment */
  printf("\n Extra Initial:%c  Age:%d
  Grade:%d",extra.initial,extra.age,extra.grade);
  return(0);
 }
```

Another example of an array of type struct using the point example created earlier follows. The rand() function is used to randomly assign values to the points coordinates. As an exercise, modify the code to handle 3-dimensional points – i.e. x,y,z coordinates

```
#include<stdio.h>
struct Point
{
   int x, y;
};

int main()
{
 int i;
 struct Point points[10];
 for(i=0;i<10;i++)
   {
    points[i].x = rand() % 100 + 1;
    points[i].y = rand() % 100 + 1;
   }

 for(i=0;i<10;i++)
 printf("\n%d %d\n", points[i].x, points[i].y);
 return(0);
 }
```

BINARY SEARCH

Suppose that you wish to look up a table and see if some record is present. For example, you may wish to search for a number x in an array of numbers called holdnum[100].

If the array is sorted and in ascending order then we can write a binary search function to find the required x. We shall set the function up to return the position in the array of the number that we are looking for and to return -1 if it is not there. In a binary search we compare x with the element located midway in the array. If x is less than this value, the search focuses on the first half of the array. If x is greater than this element, then the second half is searched. The process is repeated until the value is found or the array is empty. The sample code for this function is given below

```
int binarysearch(int holdnum[],int x,int size)
         {
          int start=0, end ,middle;
        end = size -1;
          while(start <= end)
            {
             middle =(start+end) / 2;
             if(x < holdnum[middle])end = middle -1;
             else if(x > holdnum[middle]) start = middle + 1;
           else return(middle);
          }
          return(-1);
        }
```

DYNAMIC DATA STRUCTURES

Using structures and pointers we can create useful objects such as linked lists and trees. Before looking at examples of these type of objects we need to review how to allocate memory dynamically.

DYNAMIC MEMORY ALLOCATION

Consider the task of generating an array of structures, the compiler allocates enough memory to hold the entire array. Suppose that the array can hold 50 records of employees. If only 8 or so records are entered then there is storage set aside which is never used. This is not efficient. To avoid this we make use of the C memory allocation function malloc(). Suppose that we have declared our structure data as before, to allocate enough memory for the 8 or so variables we can write the following code

```
int main()
      {
       struct data{
                   char name[50];
                   int salary;
                   } *employee;
       int j;
       for(j=0;j < 8 ;j++)
          employee =  (struct data *) malloc(sizeof(struct data) );
       return(0);
       }
```

The program above declares a structure type data that we have seen before. It than calls **malloc()** eight times. Each time **malloc()** returns a pointer to a place in memory large enough to hold a new version of the structure.

Note that as the program stands, no structure variables have been created (no names have been declared), however the function malloc() has set aside an address in memory which the program knows thus in effect eight structure variables are created. We make use of the **sizeof()** function in C to tell **malloc()** how much memory it has to set aside.

As an exercise you should use a printf() statement and the sizeof() function to see many bytes the structure takes up. For example sizeof(float) will return the value 4. The only information which has not been explained is the use of (struct data) before the malloc() function. This is an example of a cast operator.

By definition, the malloc() function returns a pointer to type char. We can `override` this by insisting on the pointer returning a pointer to a predefined data type, namely a structure of type data. As an exercise you should read up on the function **calloc()** and typecasting in C. The function **free()** is used to free up memory acquired by malloc() and calloc().

The use of dynamic memory allocation can be used in many instance for improving program memory management. This is true for character, integer, float and double arrays. The main reason for its discussion in this lesson is that a structure can become very large, often containing other structures. Think of medical records with personal details and medical history, doctor details etc. Arrays used to store all of a hospital’s medical records require a lot of memory and this is best managed dynamically.

LINKED LISTS

A linear list is a list of nodes which are arranged in a linear order. An array can be used to represent a linear list, its linear because array[i+1] is the next node past array[i] and array[i-1] is the node just before array[i].

We can also represent a linear list using pointers, this is called a linked list. In a linked list each node contains a pointer which points to the next node in the list in addition to the data in the node. At the last node in the list the pointer points to NULL ie nothing.

A linked list has an advantage over an array type list in that when an array is defined it must be of fixed size. Thus if during a program another node needs to be adding or deleted we are in trouble. A linked list, as we shall see has not got this drawback. We can use dynamic memory allocation for pointers as space is needed. Array storage is referred to as static storage.

data in node	*ptr

To represent a linked list in C we need to define a structure. The structure will contain the data and a pointer to another structure of the same type. This is called a self-referencing structure. Consider the setting up of a linked list full of numbers of type int. The following sets up the structure.

```c
struct node{
            int num;
            struct node *ptr_next;
           };
```

We want the list to be built as the numbers arrive form the keyboard. Initially the list will contain no data, thus the first node will be empty. As a new integer arrives we must allocate storage for the new node, put the number in the new node and make the new node the last node in the list. To allow this we must set up three pointers to struct node ie
struct node *first, *new, *last;

The code below is an example of how a this linked list is formed.

```c
struct node{
            int num;
            struct node *ptr_next;
           } *first,*new, *last;

int main()
{
 int value;
 first = NULL;
 if(scan("%d ",&value) !=1)  value = 0;
 while( value != 0)
    {
     first = malloc(sizeof(node) );
     first->num = value;
     first->ptr_next = NULL;
     if(first==NULL)
        first = last = new;
     else
```

```
      {
       last->next = new;
       last =new;
      }
     if(scanf(" %d ", &value) !=1) value=0;
    }
 return(0);
}
```

Note how the scanf() works in general. It returns an integer value that corresponds to the number of fields successfully scanned, converted and stored. If no fields are stored the return value is 0.

Since the linked list is initially empty, the pointer that points to the first node should point to NULL. As a new node is needed, space is set aside in memory by using the malloc() function.

The newly created node is then assigned the inputted number and this node (since this new node will be the last one) is set equal to NULL. The if -else construct is used to set the previous `last` node, to point to the new node. The new node will now of course become the last node. You should try and understand the workings of the code above and then write a function to print out the list.

This example is called a single or one way linked list because each structure contains only one link to the next node.

For a second example of a single linked list program, see the file link_list.c on Moodle.

BINARY TREES

If we wish to build a sorted list using a dynamic structure, then a two way linked list is more appropriate than a single linked list. A two way linked list is called a binary tree. This is just a structure containing two pointers to structures of the same type. A two way linked list contains three items, the data stored at a node in the list, a pointer to the previous node in the list and a pointer to the next node in the list.

<- *ptr	data at the node	*ptr ->

PROGRAMMING EXERCISES 1

1. Write struct declarations for each of the following problems
    - playing cards
    - time (hours,min,sec)
    - employees in a company
    - students in college
2. Write a program that can add, multiply and divide complex numbers. Each operation should be written as a function and a structure should be used for the complex number. (z = a + ib)
3. Write a short program which uses the structure TIME created in Q1.

4. Write a program using a structure DATE_OF_BIRTH to read in and print out a birthday.to an array of size 20.
5. Modify the program in Q4 so that 10 birthdays can be stored in an array and then printed to the screen.
6. Given that the equation of a line can be determined from its slope m and one point (x1,y2) , write a structure declaration for a line and write a function that allows the user enter a point and slope (or two points) and prints out the equation of the line.
7. Write out you own summary notes on structures in C.
8. Design first and then write a program that holds the records of ten students. The following fields are required; student number (3ditgit), student name, age and exam marks for five exams. You should store the records in an array and then write a function that searches the array for a given student no. or name. If a match is found then the student details are printed out together with the average mark. Write another function which allows the user to search for the best average mark. When found, the students details are printed out.