Lecture15
Data Types Defined by User. Structures and Unions in C Language. Arrays of Structures

Structures in C. Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it is not possible using an array. When we require using a collection of different data items of different data types we can use a **structure.** Structure in C language is another kind of composite variable. A structure is a convenient method of handling a group of related data items of different data types. A structure is a collection of variables under a single name. These variables named members or fields of a structure can be of different types, and each has a name which is used to access to it in the structure. A structure is a convenient way of grouping several pieces of related information together. A **structure** and a **union** (we will study later) can be considered as new named **data types defined by user** thus extending the number of available such named built-in-data types.

Defining a Structure. A structure type is usually defined near to the start of a program using the struct statement or the typedef statement, allowing its use throughout the program. These statements usually occur just after the #include statements in a program. Here are two examples of the struct and typedef specifications:

```
struct student
            {char name[40];
             int year;
             int clas;
             float average;
            };
struct student st1, st2 , CL[26],*ps;
```

```
typedef struct student
            {char name[40];
             int year;
             int clas;
             float average;
            } STUDENT;
STUDENT  st1, st2 , CL[26], *ps;
```

These define new data type names **struct student** and **STUDENT** and declare variables of structure type.

Accessing Members of a Structure. Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure behave just like a normal array of char, however we refer to as **st1.name.**

Here the dot is an operator which selects a member from a structure. Where we have a pointer to a structure we could dereference the pointer and then use dot as a member selector. Since accessing a member of a structure by using a pointer happens frequently, it has its own operator -> which acts as follows. Assume that **ps** is a pointer to a structure of type **STUDENT. (ps=&st1;)** We would refer to the name member as **ps->name. (*ps).name**

Structures as Function Arguments. A structure can be passed as a function argument just like any other variable. This raises a few practical issues. Where we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change. If we are only interested in one member of a structure, it is probably simpler to just pass that member. Of course if we wish to change the value of that member, we should pass a pointer to it. When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

Further Uses of Structures.

As we have seen, a structure is a good way of storing related data together. It is also a good way of representing certain types of information. Complex numbers in mathematics inhabit a two dimensional plane (stretching in real and imaginary directions). These could easily be represented here by

```
typedef struct complex

{  float real;
   float imag;
} COMPLEX;
```

Apart from holding data, structures can be used as members of other structures.
Another possibility is a structure whose fields include pointers to its own type. These can be used to build chains (programmers call these linked lists), trees or other dynamic data structures.

Arrays of structures

Arrays of structures are possible, and are a good way of storing lists of data with regular fields, such as databases. It is possible to define an array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We can define an array **S** of structures as shown in the following example.

```
typedef struct student
        { char name[40];
          int year;
          int clas;
          float average;
        } STUDENT;

STUDENT   S[100];
```

Unions in C. Unions like structures contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computers memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to observe memory. They are useful for application involving multiple members, where values need not be assigned to all the members at any one time. A union can be declared using the keyword **union** as follows:

```
union item
 { int m;
   float p;
   char c;
 } ;
union item code;
```

This declares a variable **code** of type **union.** The union contains three members each with a different data type. However we can use only one of them at a time. This is because the only one location is allocated for union variable. In effect a union creates a storage location that can be used by one of its members at a time.

Lecturer: Mihail Kulev, associate professor