Structures and Unions

Introduction

A structure stores related information about an entity. A structure is a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure.

Structure Declaration

A structure type is generally declared by using the following syntax:

```
struct structure-name {
      data_type var-name1;
      data_type var-name2;
      .........
};
```

The structure declaration, does not allocate any memory or consume storage space.

`Typedef` Declarations

`typedef` (derived from type definition) keyword enables the programmer to create a new data type name by using an existing data type. The general syntax of using `typedef` keyword is given as:

```
typedef existing_data_type new_data_type;
```

For example

```
typedef int INTEGER
```

`INTEGER` is the new name of data type `int`.

When we procede a `struct` name with the `typedef` keyword, the the `struct` becomes a new type.

```
typedef struct structure-name {
       data_type var-name1;
       data_type var-name2;
       .........
};
```

Now you can declare variables of this new data type as you declare the variables of type `int, float. char, double`, etc. To declare a variable of a structure type, write

```
structure-name var-name
```

We have not initialized `structure-name var-name`.

Initialization of Strucutres

When the user does not explicitly initialize the structure, the C automatically does it. For `int` and `float` members the values are initialized to zero, and char and string members are initialized to '\0' by default.

The initializers are enclosed in braces and are separated by commas. Initializers must match their corresponding types in the structure definition. The general syntax to initialize a strucutre variable is:

```
typedef struct structure-name {
        data_type var-name1;
        data_type var-name2;
        .........
} struct_var = { constant1, constant2, constant3, ... };
```

or

```
typedef struct structure-name {
        data_type var-name1;
        data_type var-name2;
        .........
};

struct struct_var = { constant1, constant2, constant3, ... };
```

When all members of a structure are not initialized, it is called partial initialization. In case of partial initialization, the members left unitilization the compiler will assign them their default values.

Accessing the Members of a Structure

The syntax of accessing a structure or a member of a structure is:

```
struct_var.member_name
```

To assign values to the individual data members of the structure, you may write

```
struct_var.member_name = value;
```

To input values for data members of the structure, you may write

```
scanf("%d", &struct_var.member_name)
scanf("%s", struct_var.member_name)
```

To print the values of a structure, you may write

```
printf("%s", &struct_var.member_name)
printf("%f", struct_var.member_name)
```

Memory is allocated only when we declare the variables of the structure. In the absence of any variable, structure definition is just a template that will be used to reserve memory when a variable of type `struct` is declared.

Nested Structures

A structure can be placed within another structure. The easiest way is to declare the structures separately and then group them in the higher level structure. When you do this, check that nesting must be done from inside out (from lowest level to the most inclusive).

```
typedef struct {
       data_type var-name1;
       data_type var-name2;
       .........
} struct_var1;

typedef struct {
       data_type var-name1;
       data_type var-name2;
       .........
} struct_var2;

typedef struct {
       data_type var-name1;
       data_type var-name2;
       .........
} struct_varN;
```

Arrays of Structures

The general syntax for declaring an array of structures is:

```
struct structure-name {
      data_type var-name1;
      data_type var-name2;
      data_type var-name2;
      .........
} struct struct_name struct_var[index];
```

A structure array can be declared by writing

```
struct stuct_name struct_var[index]
```

To assign values to the ith position, we write

```
struct_var.member_name = value;
```

Structures and Functions

Different ways of passing structures to functions

1. Passing individual members
2. Passing the entire structure
3. Passing the address of structure

Passing Individual Members To pass any individual member of a structure to a function, we must use the direct slection operator to refer to the individual members.

```
typedef struct {
        int x;
        int y;
} POINT;

void display(int, int) /* Display declaration */

POINT p1 = { 2, 3 }
display(p1.x, p1.y);
```

Passing the Entire Structure

We can pass an entire structure as a fuction argument. When a structure is passed as an argument, it is passed using the call by value method, i.e., a copy of each member of the structure is made.

The general syntax for passing a structure to a function and returning a structure can be given as:

```
struct struct_name func_name(struct struct_name struct_var);
```

The above syntax can vary. For example, in some situations, we may want a function to recieve a structure but return a void or the value of some other data type.

Passing Structures through Pointers

Passing large structures to functions using call by value is inefficient. It is preferred to pass structures through pointers. The syntax to declare a pointer to a structure can be given as:

```
struct struct_name {
      data_type var-name1;
      data_type var-name2;
      data_type var-name3;
      .........
} *ptr;
```

or

```
struct struct_name *ptr;
```

To access the members of a structure, we can write

/* get the structure, then select a member */

```
(*ptr).member_name;
```

We introducte the new symbol operator -¿ known as "pointing-to" operator. It can be used as:

```
ptr -> member_name;
```

Self-Referential Structures

Self-referential structures are those structures that contain a reference to the data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure. For example

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

Unions

Similar to structures, a union is a collection of variables of different data types. The only difference between a structure and a union is that in case of unions, yu can only store information in one field at any one time. When a new value is assigned to a field, the existing data is replaced with the new data.

Unions are used to save memory. They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time.

Declaring Unions

The syntax for union declaration can be given as:

```
union union_name {
     data_type var-name1;
     data_type var-name2;
     data_type var-name3;
     .........
};
```

The `typedef` keyword can be used to simplify the declaration of union variables. The most important thing to remeber about a union is that the size of a union is the size of its largesrt field.

Accessing Members of a Union

The syntax of accessing a union or a member of a union is

```
union_name.member_name
```

Initializing Unions

The difference between a structure and a union is that in case of a union, the fields share the same memory space, so new data replaces any existing data. Below is sample code:

```
typedef union POINT {
        int x;
        int y;
};

POINT p1 = { 4, 5 }  /* Illegal in case of unions */
```

Arrays of Union variables

Like structures we can also have an array of union variables. However, because of the problem of new data overwriting existing data in the other fields, the program may not display accurate results.

```
typedef union POINT {
        int x, y;
};

union POINT points[3];
points[0].x = 2;
points[0].y = 3;
points[1].x = 4;
points[1].y = 5;
points[2].x = 6;
points[2].y = 7;
```

Unions inside of Structures

Generally, unions can be very useful when declared inside a structure.

Points to Remember

• Structure is a user-defined data type that can store related information (even of different data types)

• A structure is declared using the keyworkd `struct`, followed by the structure name.

• The structure definition does not allocate any memory or consume storage space. It just gives a template that conveys to the C compiler how the structure is laid out in the memory and gives details of the member names. Like any data type, memory is allocated for the structure when we declare a variable of the structure.

• When a `struct` name is preceded with the keyword `typedef`, then the `struct` becomes a new type.

• When the user does not explicity initialize the structure, then C automatically does it. For `int` and `float` members, the values are initialized to zero and `char` and `string` members are initialized to '\0' by default.

• A structure member variable is generally accessed using a '.' (dot) operator.

• A structure can be placed within another structure. That is, a structure may contain another structure as its member. Such a structure is called a nested structure.

• Self-referential structures are those structures that contain a reference to data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure.

• A union is a collection of variables of different data types in which memory is shared among these variables. The size of a union is equal to the size of its largest member.

• The only difference between a structure and a union is that in case of unions information can only be stored in one member at a time.