C-09 Structures and Unions in C

9.1 What is a Structure?

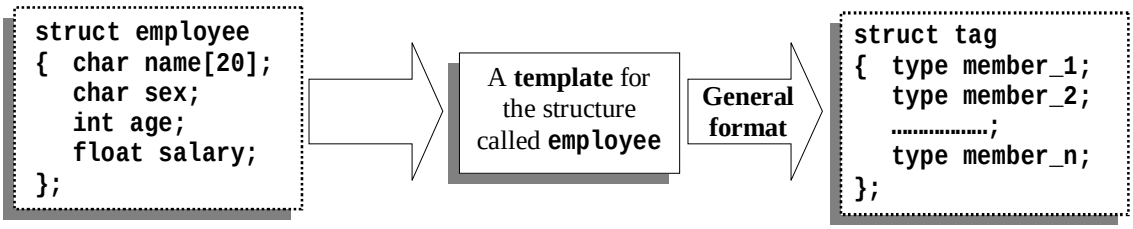
Till now we have learnt basically two types of variable declarations. One, in which a single variable is storing a single value, and a second in which a set of values of a particular type are stored in a variable. The second type is nothing but an array. We could have been satisfied with an array but the major problem with an array is that it can store values of a single type only i.e. either all elements in the array are **`int`** or **`float`** or **`char`** etc. But in general an object may need a variety of data types to describe it. For example to keep record of an employee in a company we may require the following parameters:

Name: A string type data (basically a **`char`** array) like Trinanjan Batabyal
Sex: A **`char`** type data i.e. **`M`** or **`F`**
Age: An **`int`** type data like 32
Salary: A **`float`** type data like 10500.60

Moreover it is convenient to store the above set of data under a single variable name such as **`employee`** and able to access each of the parameters of **`employee`** (such as employee_name, employee_sex, employee_age, employee_salary etc.) when necessary, by a certain convention. Also to keep a record of say 'n' number of employees in the company we can store 'n' sets of the above type of data. All these can be achieved by using a data type called a **Structure**. A **Structure** is a group of one or more variables, usually of various types, and identified by a single name.

9.2 How to Declare a Structure:

To declare a structure one has to first **define the contents** of the structure. This will then be used as a **template** to declare variables of that particular structure. We now define a structure for our above example:

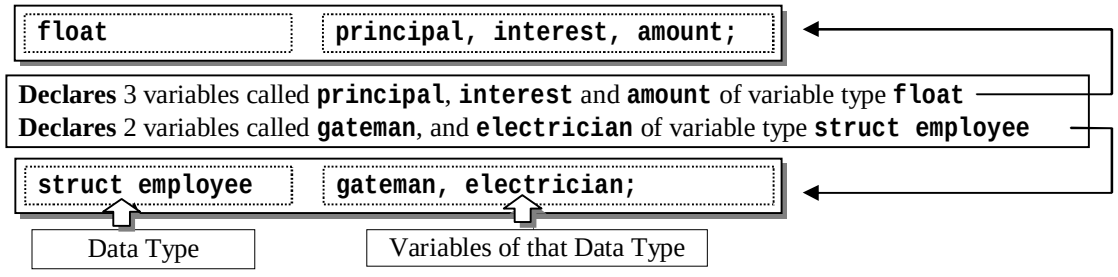


The structure name is generally called a **tag**, which specifies the particular type of structure used. The above definition indicates that we have defined a structure called **`employee`**, which requires 4 different types of data to describe it. Each **`employee`** variable will require the following sub-variables to describe it:

`char name[20];` → A character array to hold the name of the employee
`char sex;` → A **`char`** type data to hold the sex of the employee
`int age;` → An **`int`** type data to hold the age of the employee
`float salary;` → A **`float`** type data to hold the salary of the employee

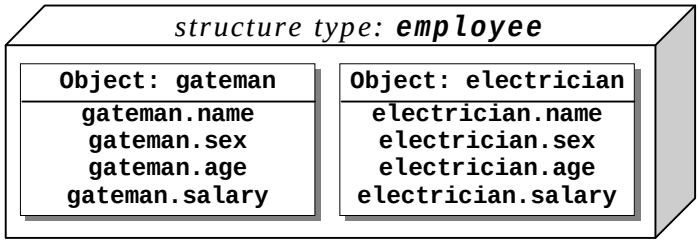
Each such **sub-variable** that makes up a structure is called a **member, element** or **field** of the structure. Usually the members of a structure are logically related i.e. they represent different components or parameters to describe a particular type of object. Note that the structure definition is terminated by a semicolon, indicating that a structure definition is a statement.

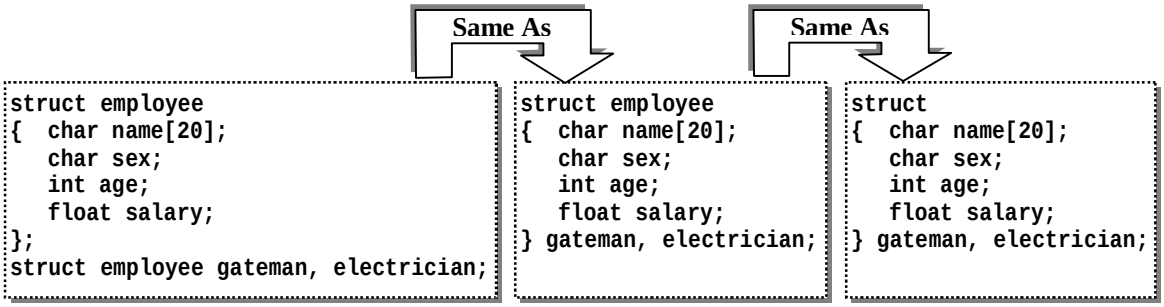
At this point no variable has actually been created but **only** the form of the data has been defined. Once the data type **`employee`** has been defined, **variables of the type `employee`** can then be declared just like any other variable declaration as shown below:



Similar to declaration of 3 **float** type variables viz. **principal**, **interest** and **amount**, the example declares 2 variables of **type** or **tag employee**. Thus **employee** describes the form or template of a structure and **gateman** and **electrician** are different objects of

the structure **employee** as shown in the figure to the right. Each of the employee objects will consist of the same set of members as defined in the template.Apart from the above declaration, the **employee** type structure variables can also be declared during the structure definition. The format shown below **combines a structure definition and variable declaration** of the same type **in a single statement**. In this case, when the structure variables are declared along with the structure definition, the **tag is optional**. Thus all the three types of **declarations shown below are same**:





Just like any other variable type, a structure itself can be a member of another structure, but **the member structure needs to be defined prior to its use in another structure**. The following example defines two types of structures. The first structure is called **date** and is used in the second structure called **product**. Finally variables of type **product** are declared.

```
struct date
{  int day;
   int month;
   int year;
};
struct product
{  char prod_brand[20];
   char model[10];
   long srl_no;
   struct date manuf_date;
   struct date sale_date;
} television, radio;
```

In the above declaration, **television** and **radio** have been declared as two **product** type structure variables, which contain the structure members **prod_brand**, **model**, **srl_no** and two **date** type structure members **manuf_date** and **sale_date**.

After the declaration of the two **employee** type variables, the compiler automatically allocates sufficient memory to hold all the elements of the structure for each of the two variables. The elements are stored in contiguous memory locations. The memory allotment is shown in a later section.

9.3 How to Enter and Read Data from a Structure:

Once a **structure** has been **defined** and variables of the same type have been **declared**, the next step is to **access** the different structure members to **input data** and subsequently to read or manipulate the data stored in the different structure variables. This operation can be achieved by using the '**dot**' i.e. '**.**' operator.

Data can be input either by **initialising** the structure variable with some initial values or by **entering the data** during runtime. Initialising a structure is similar to initialising an array. The data is put inside curly brackets separated by commas. The following example initialises a television type structure as defined:

```
struct product television = {"Samsung","Hitron",00234562, 23,4,2002, 15,6,2002};
                             prod_brand  model    srl_no    manuf_date  sale_date
```

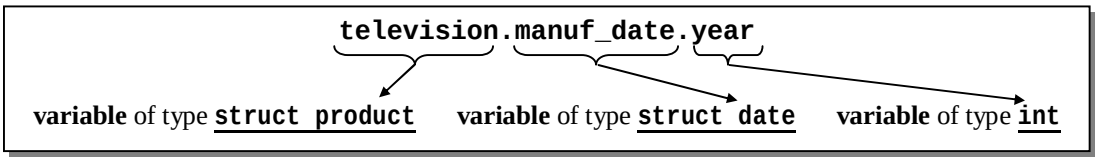
An **individual member** can be **initialised separately** also. The following statement assigns initial value of 40 to the **age** field of the **employee** type structure variable called **gateman**:

gateman.age=40; → i.e. has the general form **variable_name.member_name**

The variable or object name **gateman** followed by a period and then by the member name **age** refers to that individual member of the object.

To access the sub-member of a structure which is itself a structure, one just has to repeatedly use the 'dot' operator. Thus to assign values to different members of the **manuf_date** structure inside the structure **product** variable **television**, one has to write [Note that the **variable names** of the different structure data types have been used and **not the tags** of the different structure types as tags represent only the type of the variable like float or int]:

```
television.manuf_date.day = 23;        (NOT product.date.day = 23;)
television.manuf_date.month = 4;       (NOT product.date.month = 4;)
television.manuf_date.year = 2002;     (NOT product.date.year = 2002;)
```



Similarly **for the user to input** values into the **prod_brand** and the **day** field of **manuf_date** in the **product** structure variable **radio**, one can use the following code:

gets(radio.prod_brand); → inputs the string into the array **prod_brand** in **radio**
scanf("%d", &radio.manuf_date.day); → inputs the **day** into the variable **manuf_date** in **radio**

Similar to assigning a particular value to a member of a structure, one can also **assign** the value in a **member of one structure to another member of another structure** or a **whole structure variable to another structure variable** as shown below:

```
guarantee_card.month = television.sale_date.month
```

In the above example the right side of the assignment indicates the **month** of **sale** of the **television**. Here **television** and **sale_date** are all structure type variables while **month** is an **int** type variable. This value is assigned to the **month** member of the structure type variable **guarantee_card**. Note that in this assignment it does not matter whether the two variables **guarantee_card** and **television** are of the same structure type. What matters is that the end members (i.e. in this case **month**) should be of the same data type, which in this case are **int** type variables.

Similarly, unlike an array, **a whole structure variable can be assigned to another structure variable**. The value stored in each member of one of the structure variables will be assigned to the corresponding members of the other structure variable. But in this case the **two structures should be of the same type**.

```
television2=television1;
```

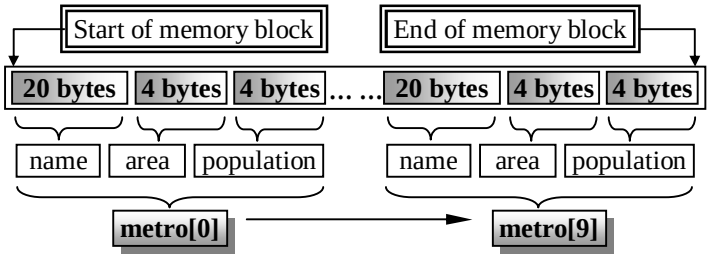
The above statement simply assigns the values in the different members of the **television1** structure variable of type **product** to a similar structure variable of type **product** called **television2**. Hence if **television1.model** is "**Hitron**", then **television2.model** will also become equal to "**Hitron**". Similarly other members of **television2** will be assigned the corresponding values of **television1**.

9.4 Array of Structures:

Just as arrays of any particular variable type can be declared, one can declare arrays that contain objects of a particular structure type variable. Thus the following declaration, declares an array variable called metro containing 10 elements of structure type city as defined below:

```
struct city
{char name[20];
 float area;
 unsigned long int population;
};
struct city metro[10];
```

The diagram to the right shows how the array elements are stored in memory for



the array **metro[10]**, each element of which is a structure of type city. It can be seen that the members of each structure are stored in **consecutive memory locations** for each of the array elements. Moreover the array elements themselves are also stored in consecutive memory location.

The example given below illustrates how to enter the array elements for the structure array **metro[]**. The program piece in reality is of little use unless the entered data is stored somewhere like a data file. We will learn about saving the entered data later. For the time being, we discuss the program piece below for academic interest only.

```
#include <stdio.h>
#define MAX 10
main()
{
  int i;
  struct city                          → Structure definition for structure type - city
  { char name[20];
    float area;
    unsigned long int population;
  };
  struct city metro[MAX];              → Array variable metro[] of type city declared
  for(i=0; i<10; i++)
   {printf("City-%d name: ",i+1);
    scanf("%s", metro[i].name);        → inputs member name to ith structure of metro[]
    printf("City-%d area: ",i+1);
    scanf("%f", &metro[i].area);       → inputs member area to ith structure of metro[]
    printf("City-%d population: ",i+1);
    scanf("%d", &metro[i].population); → inputs member population to ith structure of
   }                                     metro[]
  return (0);
}
```

First note that for a particular array element say **metro[i]**, any structure member can be accessed by referring to that particular array element followed by the structure member **separated by a 'dot'** as:

metro[i].population

array element — structure member (population) in that array element

Secondly note that in the **scanf()** function, the 'address of operator, **&**' is placed at the extreme left of the variable and **not** before any individual member of the structure. However to enter the string into the structure member **name[]** for a particular array element **metro[i]**, we have not used the '**&**' operator since the name of an array by itself, without any square brackets indicates the **address** of the first element of the array. (It is better to use the **gets()** function instead of the **scanf()** to input any string to accommodate blank spaces).

Similarly to print the **name[]** for the i^{th} member of **metro[]** we can simply write:

```
printf("City-%d name: %s",i+1, metro[i].name);
```

However to print the j^{th} character of **name[]** for the i^{th} element of the array **metro[]** we have to write:

```
printf("Character-%d of City-%d name: %c",j+1, i+1, metro[i].name[j]);
```

Note that instead of using the **%s** specifier we have used the **%c** specifier to print a character. Thus if the 3rd element of the array **metro[]** i.e. **metro[2]** has the string "**MUMBAI**" in its **name[]** field, then to print the 5th character i.e. '**A**' in the **name[]** field, we have **i=2**, **j=4** and the output will be:

```
Character-5 of City-3 name: A
```

Similar to one-dimensional arrays, two-dimensional arrays consisting of structure members can also be declared. The following example declares a structure that stores the different parameters of a CRT monitor screen pixel as a 2D matrix where the (x, y) position of a pixel is given by the values **HORI** and **VERT**:

```
struct
{  int red;
   int green;
   int blue;
   int hue;
   int saturation;
   int luminosity;
```

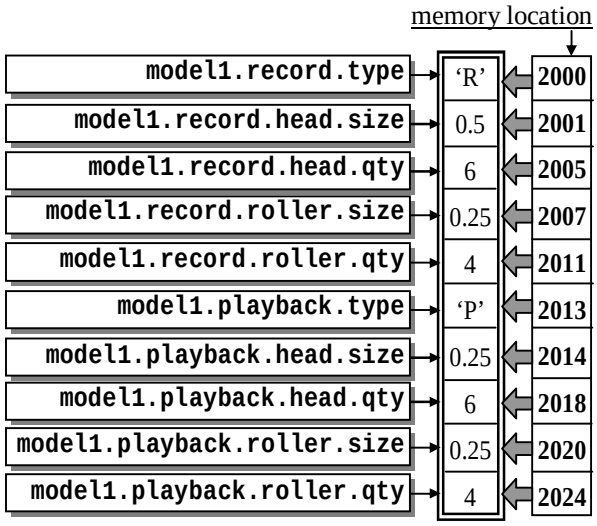
```
} pixel[HORI][VERT];
```

9.5 Passing Structure Members to Functions:

Just like any variable that can be passed to a function, we can pass either a structure member or an entire structure to a function.

As a structure **member** behaves like any other variable, we can follow the same procedures to pass them to functions as we do with other variables. The following program uses a function to compare the stock of screws for recorders in a factory against the number of screws required to assemble a particular recorder unit. The variable **model1** is a structure type variable and has structure member like **deck**, which is in turn having structure member like **screw**. The variable **model1** has been initialised with the values of the different member quantities. Hence the quantity of **screw** for **playback head** is given by:

```
model1.playback.head.qty = 6;
```




```
#include <stdio.h>
void screw_qty(int unit, int stock);
int main()
{ int play, rec;
  struct screw
  { float size;
    int qty; };
  struct deck
  { char type;
    struct screw head;
    struct screw roller;};
  struct recorder
  { struct deck record;
    struct deck playback;};
  struct recorder model1 = {'R', 0.5, 6, 0.25, 4, 'P', 0.25, 6, 0.25, 4};
  printf("\nEnter existing stock of playback-head-screw for Model-1: ");
  scanf("%d", &play);
  screw_qty( model1.playback.head.qty, play );
  printf("\n\nEnter existing stock of recording-head-screw for Model-1: ");
  scanf("%d", &rec);
  screw_qty( model1.record.head.qty, rec );
  return (0);
}
void screw_qty(int unit, int stock)
{ int purchase;
  purchase=25*unit-stock;
  if(purchase>0)
    printf("\nThe total number of screws required = %d", purchase);
  else
    printf("\nStock is sufficient");
}
```

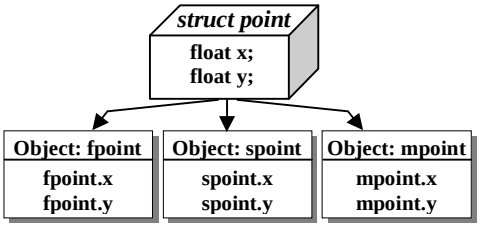
Output of the above program:

```
Enter existing stock of playback-head-screw for Model-1: 60
The total number of screws required = 90
Enter existing stock of recording-head-screw for Model-1: 400
Stock is sufficient
```

9.6 Passing Entire Structures to Functions:

The next problem is used to pass an entire structure to a function. **An entire structure can be passed to a function using the normal call by value method**. In the next section we will discuss the call by reference method using pointers.

The following example uses a structures called **struct point** to denote the pair of (x,y) co-ordinates of any point and finds the co-ordinates of the middle point of a line segment represented by the two points (x1,y1) and (x2,y2):




```
#include<stdio.h>
struct point
{  float x;
   float y;
};
struct point MakePoint( float a, float b );
struct point MiddlePoint( struct point p1, struct point p2 );
main()
{  struct point fpoint, spoint, mpoint;
   float x1, x2, y1, y2;
   printf("\n### PROGRAM TO FIND THE MID POINT OF A LINE SEGMENT ###\n");
   printf("Enter the x co-ordinate of point1: "); scanf("%f", &x1);
   printf("Enter the y co-ordinate of point1: "); scanf("%f", &y1);
   printf("Enter the x co-ordinate of point2: "); scanf("%f", &x2);
   printf("Enter the y co-ordinate of point2: "); scanf("%f", &y2);
   fpoint = MakePoint(x1,y1);
   spoint = MakePoint(x2,y2);
   mpoint = MiddlePoint( fpoint, spoint );
   printf("\nThe x co-ordinate of the mid point = %.2f", mpoint.x );
   printf("\nThe y co-ordinate of the mid point = %.2f", mpoint.y );
   return(0);
}
struct point MakePoint( float a, float b )
{  struct point temp;
   temp.x = a;
   temp.y = b;
   return temp;
}
struct point MiddlePoint( struct point p1, struct point p2 )
{  struct point temp;
   temp.x = (p1.x + p2.x)/2.0;
   temp.y = (p1.y + p2.y)/2.0;
   return temp;
}
```

Output Of The Program:

```
   ### PROGRAM TO FIND THE MID POINT OF A LINE SEGMENT ###
   Enter the x co-ordinate of point1: 5.6
   Enter the y co-ordinate of point1: 2.2
   Enter the x co-ordinate of point2: 3.6
   Enter the y co-ordinate of point2: 7.4
   The x co-ordinate of the mid point = 4.60
   The y co-ordinate of the mid point = 4.80
```

The above program uses a structure data type called **point**, which stores the **x** and **y** co-ordinates of a point. Note that the structure is defined outside the **main()** function, so that it becomes a **global variable type** and can be accessed by all the functions. Within the **main()** function, three variables are declared of type **struct point**, called **fpoint**, **spoint**, and **mpoint**. Then the **x** and **y** members of **fpoint** and **spoint** are input from the user. The function **MakePoint()** is then called twice with the above **x** and **y** co-ordinates as arguments. The function returns a **struct point** type data, which is assigned to **fpoint** and **spoint** respectively for each function call. Finally the function **MiddlePoint()** is called with the structure type arguments **fpoint** and **spoint**. The function **MiddlePoint()** uses a temporary **struct point** type variable called **temp**, to calculate the co-ordinates of the middle point. Within the function header, the formal arguments **p1** and **p2** of type **struct point** are used to receive the actual arguments **fpoint** and **spoint**. The components of the middle point are calculated within the function and returned as a **struct point** type data, to be received by the variable **mpoint** in **main()**. Finally the **x** and **y** components of **mpoint** are displayed in **main()** as **mpoint.x** and **mpoint.y**.

9.7 Addressing Structures by using Pointers:

One can declare a pointer to a structure variable in the same manner one declares a pointer to any other variable. To declare a pointer to a structure variable of type point, the syntax is:

```
struct point
{  float x;
   float y;
}  point1, point2;                    → Variables point1 and point2 of type point declared

struct point *p_point1, *p_point2;    → Pointer variables p_point1 etc. of type point declared

p_point1=&point1;                     → Address of point1 assigned to pointer p_point1
p_point2=&point2;                     → Address of point2 assigned to pointer p_point2
```

In the above example, two structure type variables **point1** and **point2** are declared. Next, two pointers of type **struct point** are declared called **p_point1** and p_**point2** and the addresses of the variables **point1** and **point2** are assigned to them.

We now will access the **structure members using the pointers** that are pointing to the addresses of the structure variables and assign values to those memory locations.

```
(*p_point1).x = 1.2;
(*p_point1).y = 5.5;
```

The dereferencing operator '*****' in this case indicates the **contents of the address pointed to by the pointer p_point1** and the address in this case contains the structure variable **point1**. Thus one can say that:

(*p_point1) is **equivalent to point1**

Therefore the meaning of the above statement is that we are referring to the member 'x' of the structure variable pointed to by the pointer **p_point1** i.e.

```
(*p_point1).x = 1.2;      same as      point1.x = 1.2;
(*p_point1).y = 5.5;      same as      point1.y = 5.5;
```

However instead of writing **(*p_point1).x** if we had written ***p_point1.x** without the brackets, it would have been wrong. This is because the dot operator '**.**' has a higher priority than the indirection operator * and accordingly the above statement would have been interpreted as ***(p_point1.x)** which is incorrect, as in this case the dereference operator * instead of working on **p_point1** is working on the parameter **x**, which is not a pointer type variable.

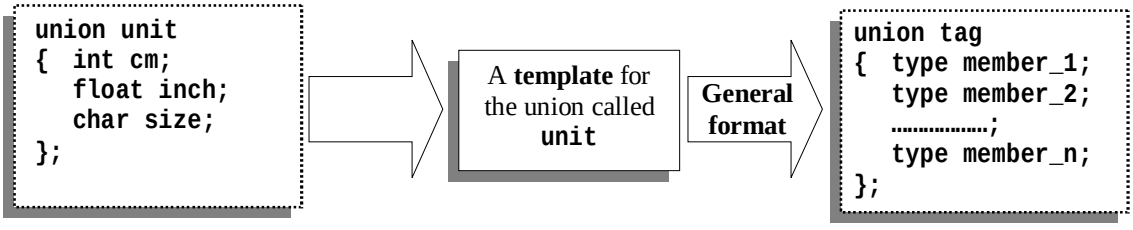
To avoid this confusing notation C provides us with the arrow operator given by '**->**' (a hyphen followed by the greater than sign). Using the arrow operator and the pointer to the structure, one can refer to the members of a structure. Therefore:

```
p_point1-> x = 1.2;   same as   (*p_point1).x = 1.2;   same as   point1.x = 1.2;
p_point1-> y = 5.5;   same as   (*p_point1).y = 5.5;   same as   point1.y = 5.5;
```

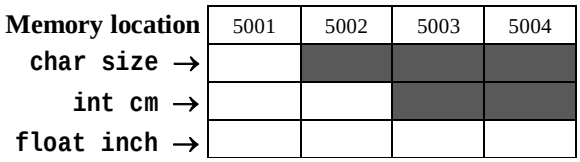
Just replace the structure variable name with the corresponding pointer to the structure variable and replace the dot with the arrow.

9.8 Union:

A union is another user defined composite data type similar to a structure and defined using the keyword **union**. However the major difference between a union and a structure is that unlike the members of a structure which occupy different storage locations, the **members of a union share the same storage area** within the memory. Thus it can handle only one member at a time and is basically used to conserve memory. The syntax of a union is similar to that of a structure and is shown below:



```
union unit
 { int cm;
   float inch;
   char size;
 } Tshirt;
```



All union members share the same memory locations

The above definition defines a union with tag **`unit`**. The members include an **`integer`** variable called **`cm`**, a **`float`** variable called **`inch`** and a **`character`** variable called **`size`**. The union unit is used to describe a shirt size in different units. Next a variable called **`Tshirt`** of type **`union unit`** is declared. We can see from the adjacent diagram that the member variables of **`union unit`** are of different data types. Hence the compiler allocates enough space to hold the largest member of the **`union`**. In this case, it is the **`float`** member called **`inch`** which occupies 4 bytes and hence the maximum space allotted is 4 bytes. All the three members share the same memory location starting from 5001 to 5004. However the character variable requires only one byte, and the integer only two bytes.

To access a **`union`** member, one can use the same syntax as one uses to access a structure member as shown below:

```
Tshirt.cm = 40;
Tshirt.inch = 15.75;
Tshirt.size = 'L';
```

The above three statements assign different values to the different member variables. Note that the different values corresponding to a shirt size of 40cm are assigned to the different members. However, since at a time only one of the member variables can be active, one cannot access more than one member variable at any instance of time. This is the major difference between a structure and a union. One can initialise all structure members, but **one can initialise only a single union member**. Thus the following set of statements will produce garbage values.

```
printf("\nEnter shirt size in cm");
scanf("%d", &Tshirt.cm);
if(Tshirt.cm == 40)
   Tshirt.inch = 15.75;
printf("\nEntered shirt size = %d", Tshirt.cm);
```

In the above program piece, the value is input into the member variable **`Tshirt.cm`**. Next, in case the **`if`** statement is correct, the **`Tshirt.inch`** member variable is assigned the value **`15.75`**. This automatically erases the value of **`40`** from the shared memory location for **`Tshirt.cm`**. Hence when in the last statement **`Tshirt.cm`** is asked to be printed, it will print some garbage value as no value exists for **`Tshirt.cm`** at that instant.

Like nested structures, one can have unions with union members, structures with union members and unions with structure members.

```
union type
 { char colour[12];
   int size;}

Struct clothes
 { char manufacturer[20];
   float cost;
   union type description;} shirt, Tshirt;
```