**Structures and Related**

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 is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types.

**Example:**struct lib\_books   
{   
char title[20];   
char author[15];   
int pages;   
float price;   
}books;

**With Function passing structures members Passing Entire structure to function**

struct employee{

int emp\_id;

char name[25];

char department[10];

float salary; };

void main() {

static struct employee emp1= {

12,

“sadanand”,

“computer”,

7500.00 };

/\*sending entire employee structure\*/

display(emp1); }

/\*function to pass entire structure variable\*/

display(struct employee empf ) {

printf(“%d%s,%s,%f”, empf.empid,empf.name,empf.department,empf.salary); }

void main()   
{   
struct emp{

int emp\_id;   
char name[25];   
char department[10];   
float salary;   
}; struct emp1={125,”sampath”,”operator”,7500.00};

/\* pass only emp\_id and name to display function\*/   
display(emp1.emp\_id,emp1.name);

**}**

/\* function to display structure variables\*/   
display(int e\_no,char e\_name[])   
{   
printf(“%d%s”,e\_no,e\_name);

}

**Array of structures**

struct inventory {

int part\_no;

float cost;

float price;

};

struct inventory table[4];

which defines an array with four elements, each of which is of type struct inventory, i.e. each is an inventory structure.

**Pointers and structures**

One can use structures in the same ways as other basic data types. This includes making pointers to structures. With pointers, though, in accessing structure members an *arrow* notation is used, rather than the *dot* notation:

#include <stdio.h>

typedef struct {

int n, d;

} frac;

int main(void) {

frac \*f1;

f1 = (frac \*) malloc(sizeof(frac));

f1->n = 1;

f1->d = 3;

printf("The fraction is %d / %d\n", f1->n, f1->d);

return 0;

}

**If two pointers to structures are there like**

**Struct hi \*p1;**

**Struct bye \*p2;**

**So we can use**

\*p1 = \*p2 //would copy the structure pointed to by p2 to the structure pointed to by p1

p1 = p2 //doubt

**Nested structure**

Structures can contain other structures as members; in other words, structures can nest. Consider the following two structure types:

struct first\_structure\_type {

int integer\_member;

float float\_member;

};

struct second\_structure\_type {

double double\_member;

struct first\_structure\_type struct\_member;};

struct second\_structure\_type demo;

demo.double\_member = 12345.6789;

demo.struct\_member.integer\_member = 5;

demo.struct\_member.float\_member = 1023.17;

**Self Referential Structure**

It contains a pointer to that structure that either points to the structure itself either directly or indirectly through a chain of pointers. A self referential structure is used to create data structures like linked lists, stacks, etc. A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

typedef struct list\_node {  
void \*data;  
struct list\_node \*next;  
} linked\_list;

In the above example, the list node is a self-referential structure – because the \*next is of the type struct list\_node. More clearly i.e

struct list{

int value;

struct list \*next;

};

struct list \*first = (struct list \*)malloc(sizeof(struct list)); //pointer to the first element in the list

first = first->next; //set the pointer to the second element in the list, using pointer in first node

**Note:**If it is not a self referential also, and if a pointer to structure is declared like

Struct list{

Struct emp \*emp1;}//In this case also we have to create memory allocation for emp1 structure to access the members of structure emp as above otherwise we cannot.

**Structure Padding**

Structure padding is adding extra bits at the end of the structue, so that the structure completes the word boundary.

**How structure padding done**

Structure Padding is done to do memory alignment. As size of pointer is 4 bytes, if everything is organized by multiple of 4, that it will be easier and faster to calculate the address and processing them. As structures are used to club different size variables, compiler will align to 4 byte boundaries and for that it needs padding.   
  
Structure padding is done by the compilers and this depends on the architectures. Some architectures cannot access the data which will be stored on the odd addresses or they may find difficult to access it. This is the reason for padding of extra bytes.   
  
Padding will be done by compiler to structure’s members and to the structure as a whole also. Compiler pads structure as whole because this allows each member of structure aligned in array of structures.

Most processors require specific memory alignment on variables certain types. Normally the minimum alignment is the size of the basic type in question, fo instance this is common  
  
**char variables** can be byte aligned and appear at any byte boundary  
  
**short (2 byte)** variables must be 2 byte aligned, they can appear at any even byte boundary. This means that 0x10004567 is not a valid location for a short variable but 0x10004566 is.  
  
**long (4 byte)** variables must be 4 byte aligned, they can only appear at byte boundaries that are a multiple of 4 bytes. This means that 0x10004566 is not a valid location for a long variable but 0x10004568 is.  
  
Structure padding occurs because the members of the structure must appear at the correct byte boundary, to achieve this the compiler puts in padding bytes (or bits if bit fields are in use) so that the structure members appear in the correct location. Additionally the size of the structure must be such that in an array of the structures all the structures are correctly aligned in memory so there may be padding bytes at the end of the structure too.

Example 1

Struct EX  
{  
Char a;  
Short int b;  
Char c;  
Long d;  
};  
  
Due to structure padding this structure size is 12 bytes (a=1 + padding=2 + b=2 + c=1 padding=3 + d=4) insted of 8 bytes (1+2+1+4).

To avoid structure padding, we can use **pragma** & **structure packing**

**Structure packing**

The packed attribute specifies that a variable or structure field should have the smallest possible alignment one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute.

Ex 1 Ex 2

struct my\_struct{

char c;

int a;

char b;

}\_\_attribute\_\_((packed));

Output : 6 bytes

struct foo  
{  
char a;  
int x[2] \_\_attribute\_\_ ((packed));  
};

So with packing, it will show the size of structure as 9 bytes otherwise 12 bytes

**Pragma**

Pragmas often control actions of the compiler and linker. A pragma always begins with a number sign (#).by using #pragma you can avoid structure padding. and that to you can use it in linux or unix. So pragma is to change change byte alignment.

Ex

struct \_s1{

unsigned int i;

unsigned char c;

unsigned long a;

unsigned short e;

} s1;

.

Size of this above structure should of 11 Bytes. But due to default byte alignment (8 byte/4 byte) which is different for different compilers. The size of structure is of 16 Bytes. In order to change the alignment, we will have to do something like this using pragma

#pragma pack(push,1) //Changing for current program

typedef struct \_s1{

unsigned int i;

unsigned char c;

unsigned long a;

unsigned short e;

//unsigned char b;

} s1;

#pragma pack(pop) //Changing back to compiler alignment

This will change the byte alignment to 1 Byte and thus size of structure will be exactly 11 bytes. We can use instead of #pragma pack(push,1) in two lines i.e #pragma pack(push)

#pragma pack(1)

**Union**

A union, is a collection of variables of different types, just like a structure. However, with unions, you can only store information in one field at any one time.   
  
You can picture a union as like a chunk of memory that is used to store variables of different types. Once a new value is assigned to a field, the existing data is wiped over with the new data.   
  
A union can also be viewed as a variable type that can contain many different variables (like a structure), but only actually holds one of them at a time (not like a structure). This can save memory if you have a group of data where only one of the types is used at a time. The size of a union is equal to the size of it's largest data member. In other words, the C compiler allocates just enough space for the largest member. This is because only one member can be used at a time, so the size of the largest, is the most you will need. Here is an example:

union time  
{

long simpleDate;  
double perciseDate;

}mytime;

The union above could be used to either store the current time (in seconds) to hold time accurate to a second. Or it could be used to hold time accurate to a millisecond. Presumably there are times when you would want one or the other, but not both. This declaration should look familiar. It is the same as a struct definition, but with the keyword union instead of struct.

**Difference between structures**

* Union allocates the memory equal to the maximum memory required by the member of the union but structure allocates the memory equal to the total memory required by the members.
* In union, one block is used by all the member of the union but in case of structure, each member have their own memory space.
* Union is best in the environment where memory is less as it shares the memory allocated. But structure cannot implement in shared memory.
* As memory is shared, ambiguity is more in union, but less in structure.
* Self referential union cannot be implemented in any data structure, but self referential structure can be implemented.

Example

union foo {

int a;   // can't use both a and b at once

char b;

} foo;

struct bar {

int a;   // can use both a and b simultaneously i.e int a,b

char b;

} bar;

union foo x;

x.a = 3; // OK

x.b = 'c'; // NO! this affects the value of x.a!

struct bar y;

y.a = 3; // OK

y.b = 'c'; // OK

**Note**: We can have a struct inside a union and vice versa also.And also we can define a structure or union that contains, as fields, structures and unions without names i.e

struct {

int a;

union {

int b;

float c;

};

int d;

} foo;

In this example, the user would be able to access members of the unnamed union with code like foo.b. Note that only unnamed structs and unions are allowed, you may not have, for example, an unnamed int. You must never create such structures that cause ambiguous field definitions. For example, this structure:

struct {

int a;

struct {

int a;

};

} foo;

It is ambiguous which a is being referred to with foo.a. Such constructs are not supported and must be avoided. In the future, such constructs may be detected and treated as compilation errors.

**Real time example for Union**

Still has to get