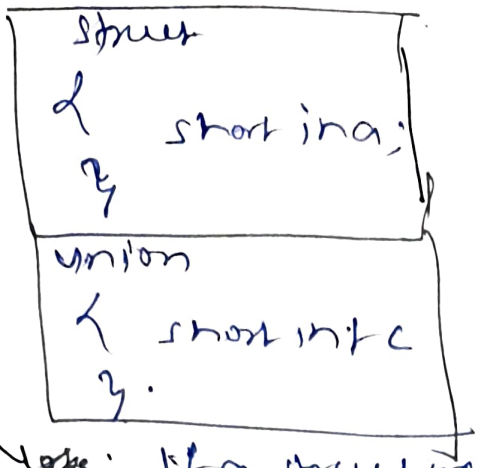


Note: If a structure contains the anonymous structure or union then the members of anonymous structure or union are considered to be members of structure or union

Ex:

```
struct mystruct
{
    union
    {
        short int a;
        short int b;
    };
    short int c;
};
struct mystruct md;
md.a = 10
md.b = 20
md.c = 30
```

} valid.

ex:2

```
struct mystruct
{
    union
    {
        struct
        {
            short int a;
        } data;
        struct
        {
            short int b;
        };
    };
```

short int c;

};

struct mydata md;

md.a=10; // Invalid

~~md.data=20;~~

md.data.a=10 // valid

md.b=20 valid

md.c=30 valid

Flexible array member in C

- introduced in C 9.9
- enables user to create an empty array in a structure. and size of empty array can be changed at runtime.
- this empty must be declared as the last member of the structure. and structure must contain atleast one or more named members.

* Flexible array A.T. C99, as a special case the last element of the structure with more than one named memb. may have incomplete type. which is called flexible array member

Note: when structure has flexible member then the entire structure becomes incomplete type. an incomplete type struct is structure that has less of info. about its members.

Incomplete type:

(1) an array type whose dimension not specified

(2) a struct or union whose members not specified.

ex: struct stud
{ int roll
char name[]; // incomplete array as last member
};

How to use flexible array

int n=5;
struct stud *ptr = malloc (sizeof(struct stud) + sizeof (char[5]));

i.e which is same as below

struct stud
{ int roll;
char name[5];
} *ptr;

what must be the sizeof structure that has flexible array members?

struct stud
{ int roll;
int class;
char name[]; // A
};

→ size will be 8 bytes we should ignore flexible array member while calculating the sizeof struct.

struct stud s1

Why does a flexible array require dynamic memory

(29) struct hack in C

- This technique permits user to create a variable length member in the structure.
- When we create 0 size array then the structure becomes incomplete.
- When we create an incomplete type member in the structure, then the structure becomes incomplete. and this technique is called struct hack.

ex:
```
struct stud
{   int roll;
    int class;
    char name[0];
};
```

structure bit fields: It allows packing of data in a structure or union and prevents wastage of memory

```
struct stud
{   int data1 : 1
    int data2 : 2
    int data3 : 3
};
```
// the layout of bit field is implementation defined

Polymorphism in C with the help of function pointer

structure with some function ptr. represents generic containers and the function pointer behaves like C++ member function

ex:

```
typedef struct
{
    int (*OpenSocket) (void)
    int (*CloseSocket) (void)
    int (*ReadFromServer) (int, char*, int)
    int (*WriteToServer) (int, char*, int)
} sCommStruct;
```

→ generic communication for server comm

for tcp/ip: initialize function pointer with tcp/ip related api and for udp we should initialize them with udp api

How to use above struct

```
sCommStruct *sComm = malloc(sizeof(
                     sCommStruct));
```

```
sComm → OpenSocket = &TcpSocketOpen;
sComm → CloseSocket = &TcpCloseSocket;
sComm → ReadFromServer = &TcpSocketReceive
sComm → WriteToServer = &TcpSocketWrite
```

calling

```
int result = sComm → OpenSocket()
```

(31)

memset in C

- used to fill block of memory with a particular value.

```
void *memset(void *ptr, int x, size_t n);
```

- ptr = starting add of mem to be filled
- x value to be filled
- n = no of bytes to be filled starting from ptr to be filled.

```
char name[50] = "Rajkalash Raykar";
memset(name,'.', 8)
```

O/P: Rajkalash.....Raykar

- if ptr is of char type then O/p is defined
  otherwise O/p is not defined

*
```
int arr[10];
memset(arr, 10, 10*sizeof(int));
```

O/P: undefined not predictable: since memset works character by character.

Structure padding in C

* memory alignment affects the performance of the CPU. And the processor takes some extra time to access the unaligned mem
  → alignment is the micro optimization technique.

* 32 bit processor, word size 4

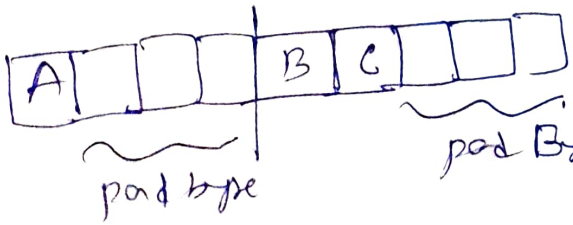
Note: Generally compiler handles the concept of alignment and it aligned the variable in their boundary.

* When we create the structure or union then the compiler inserts some extra bytes (padding) between the member of the structure or union for alignment.

* padding increases the performance of the processor at the cost of memory. In structure or union data is aligned as per the size of the highest bytes member to prevent the penalty of performance.

Ex:

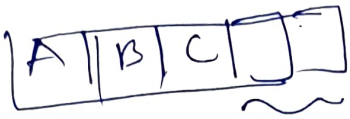
```
struct stud
{
    char A
    int B
    char C
};
```



total 6 bytes padding.

(33) struct stud

```
{
    int A;
    char B;
    char C;
};
```



2 byte padding

OR

struct stud

```
{
    char A;
    char B;
    int C;
};
```

2 byte of padding

Note: In the case of structure or union we can save the wastage of mem to rearrange the structure mem in the order of largest size to smallest size or vice versa.

How to avoid structure padding

(i) By using #pragma pack(1)

(ii) attribute (__'attribute__((__packed__))).

Ex: (i) #pragma pack(push, 1)

typedef struct

```
{
    double A
    char B
    char C
} stud;
```

#pragma pack(pop)

It tells that the alignment is 1 byte so no padding

ex: #pragma pack(push, 2)

struct stud

{ double A;
  char B;
  char C;
};

#pragma pack(pop)

alignment is 2 byte

2 byte no padding

ex-3

#pragma pack(push, 4)

typedef struct

{ double A;
  char B;
  char C;
} student;

alignment 4 byte

2 byte for padding

ex-4 Default alignment

typedef struct

{ double A;
  char B;
  char C;
} stud;

alignment = 8 byte

& 6 byte for padding

How to find size of structure without using sizeof operator?

ex:

typedef struct

{ double A;
  char B;
  char C;

(35) } student;

student myInfo[2] = {0};

student *ptr = NULL

int sizeOfStruct = (char*)(ptr+1) - (char*)ptr

Or

ptr++ will also give sizeof increases the block ptr by size of block.

what struct hack

- used to create flexible length member.
- by creating a zero size array inside structure. ↓ to dynamically assigned mem.

why not use pointer instead of struct hack

* extra space for pointer + data will be some where else it will not be completely packed inside structure ~~at the same~~ place.

ex. struct myData

{ int roll;
int class;
char name[0]; while calculating the sizeof struct hack we should ignore 0 size array

};

pointer inside structure

```
typedef struct
{
    int iAge;
    char *pcName;
} Info;

Info myInfo = {20, "Ray"};

myInfo.pcName
```

pointer to structure:

```
typedef struct
{
    int *piAge;
    char *pcName;
} Info

Info *ptrInfo = NULL;

ptrInfo = malloc(sizeof(struct Info));
ptrInfo->piAge = malloc(sizeof(int));
ptrInfo->pcName = malloc(sizeof(
                  4 * or sizeof(char));
```

(27) Array of structure

```
struct Employee
{
    int eid;
    int salary;
    char name[10];
};
```

precedence of [] and (.) dot is same and associativity is from left to right

```
struct Employee e[10];
```

```
e[0].eid = 1001
e[0].salary = 20000
strcpy(e[0].name, "Rajkalash");
```

Union — main purpose is to save mem.

Used to store different data types in the same memory location which means an efficient way of using the mem but not at the same time.

union declaration, union tag is optional and each member of union define as normal

```
union myData
{
    int Age;
    float fees;
    char name[4];
} data // optional
```

→ union variable declaration

```
union myData
{
    int Age;
    float fees;
    char name[4];
};
union myData m1, *ptr;
```

(*ptr).Age
or
ptr->Age

(37) Array of structure

```
struct Employee
{   int eid;
    int salary;
    char name[10];
};
```

~~Precedence~~ of [] and . is same and associativity is from left to right

```
struct Employee e[10];
```

```
e[0].eid=1001
e[0].salary=20000
strcpy(e[0].name, "Rajkailash");
```

Union — main purpose is to save mem.

Used to store different data types in the same memory location which means an efficient way of using the mem but not at the same time.

allocate common memory for overlapping use.

union declaration, union tag is optional and each member of union define as normal

```
union myData
{   int Age;
    float fees;
    char name[4];
} data // optional
```

→ union variable declaration

```
union myData
{   int Age;
    float fees;
    char name[4];
} m1, *ptr;
```

(*ptr).Age
or
ptr->Age

Note: If a union has two or more than two types of member and assigned the value to anyone member, but he accessed the value through other type members. Then the result will be unreliable.

```
union Test
{
    float fAmount;
    int iAmount;
};

union Test d1;

d1.fAmount = 204.50;
printf("%d", iAmount);
```

unreliable.

* size of union is same as size of largest member + required padding.

* Anonymous union or structure:

without name

```
struct Empty
{
};
```

// size 0 → gcc permits

* Important points

* we can create bit field using union.

```
union data
{
    unsigned int a:4
    unsigned int b:4
    int c
};
```

(39) * each non bit field member of a structure or union object is aligned in an implementation defined manner appropriate to its type

* Designated initialization of union is supported in C99 and C11

```
union Data
{
    int a;
};
```

union Data d1={.a=4};

* flexible array is not supported in union

Bit field in C

The order of allocation of bit field is implementation defined.

```
struct BitField
{
    int data1 : 1
    int data2 : 2
    int data3 : 3
};
```

(10) we cannot create pointer to bitfield and also we can use address of (&) to the bit field member.

& struct BitField data

& data.data1 ✗ illegal

(11) we cannot create an array of bitfield.

```
struct Data
{
    int a : 1
    int b : 2
};
```

int c[5] : 5 ✗ illegal

(40) (iii) The bit field must also be long enough to contain the bit pattern.

```
struct Data
{ unsigned int a:2;
  short b:17 //illegal
}
```

(iv) The alignment of the addressable storage unit is unspecified.

(v) The bit field declarations with no declarator, is called an unnamed bit field. If the width of unnamed bitfield is 0. it indicates that no further bitfield is to be packed. into the unit in which the previous bitfield if any was placed.

ex.

```
typedef struct
{ unsigned int data1:5;
  unsigned int data2:8;
} Data
```

~~struct~~

Data d1;

sizeof(d1) = 4

```
typedef struct
{ unsigned int a:5
  unsigned int :0; //unnamed bit field
  unsigned int c:8
} UnnamedBitfield;
```

(v) Unnamed bitfield unbf;

sizeof(unbf) = 8

(vi) we cannot calculate the size of bitfield in C using sizeof operator

~~struct Bit~~

typedef struct

{ int a:5;
  int b:2;
  int c:8;
} Bitfield;

Bitfield bf;

bf sizeof(bf.a) // illegal.

Void pointer in C

- called a generic pointer and has no associated data type. It can store the add of any type of object and it can be type casted to any type
- A.T.C. standard. the pointer to void shall have the same representation and alignment as pointer to char type

void *ptrName

void *ptrHandle

What is the size of void pointer?

- Its size is same as character pointer.

Dereferencing void pointer

For void pointer we cannot use (*) indirection operator directly.. before using * to void Ptr we have to typecast it, it enables the compiler to predict the Data types

ex: void *pvData;
int a=10;
pvData= &a
*pvData ✗ illegal

typecasting to int *(int*) pvData

why void pointers are used?
It is used for code reusability.
To store the address of any object and whenever required we can get back the object through the indirection operator through proper casting

Arithmetic operation void pointer

int iData[3]={100, 200, 300}
void *pvData;
pvData= &iData[0];
pvData+1 incorrect

(int*)(pvData)+1 correct

(43) Note: A.T.C. standard arithmetic opr on void pointer is illegal that means C standard does not allow pointer arithmetic with void pointer. However in GNU C, addition and subtraction opn are supported. Or void pointer to assume the size of void is 1.

Ex: int ipdata[] = {100, 200, 300};

void * pvdata = &ipdata[0]

pvdata = pvdata + 1

O/P: (int*) pvdata

O/P: 300 or compile error

Advantage

- We can create generic fn using void ptr. Ex: memcpy and memmove library fn
- can be converted to any other data type malloc, calloc, realloc

NULL pointer

A.T.C standard an integer constant exp with the value 0 or such an exp. cast to type void* is called null ptr constant. If a null ptr constant is converted to pointer type then the resulting pointer is called null ptr

#define NULL ((void*)0)

Note: The behaviour of null pointer is defined by C standard, if we try to dereference the NULL ptr we will get segmentation fault.

(44) NULL ptr :→ prevents the ptr to become wild pointer

* A.T.C 0 is a also new null ptr constant

Dangling pointer

Can we use sizeof operator on NULL ptr?

Yes. The result is similar to other pointers which means if pointer size of platform is 4 Byte. then size of operator on NULL gives o/p 4 bytes.

it is *ptr=0 int *ptr=0

wild pointers

Uninitialized pointers are known as. The beha viour of wild pointers is undefined since they point to some arbitary memloc.

int *ptr; wild pointer

Dangling pointer:

when referencing objects deleted or deallocated some dangling ptr the value of pointers

(I) when pointer variable goes out of scope
(II) After destroying the stack frame
(III) Deleting the memory explicitly.

Ex. int *ptr = (int*) malloc (sizeof(int) * sizeof(int))

free(ptr) // now ptr is dangling ptr.

ptr=NULL ptr is no more dangling ptr.

(15) Difference betn ++*p, *p++ and *++p

(i) precedence of postfix (++) higher than prefix ++

(ii) associativity of ++ is left to right

(iii) " prefix ++ right to left

(iv) precedence of prefix (++) and * same and left to right associativity

(v) precedence of postfix (++) is higher than * and associativity is also diff

precedence (postfix ++) > prefix ++ > *

(preced) of prefix = (preced) of *

(1) Ex: int a[] = {100, 200, 300};
int *p = arr

++*p ≈ ++(*p) O/P = 101

*p++ ≈ *(p++) O/P = 200

*++p ≈ *(++p) O/P = 200

Dynamically allocate 2D array

int *pBuffer = malloc(7 * sizeof(int))

int **ptr = (int **) malloc(7 * sizeof(int *))

2D array array

1. By using pointer to pointer

int **ptr = NULL

ptr = (int **) malloc(7 * sizeof(int *))

```
for(int i=0; i<7; i++)
    ptr[i] = (int*) malloc(sizeof(
                int));
```

[j x 5]

using single pointer

```
int *ptr = NULL;
int row, col;
ptr = (int*) malloc(row*col*sizeof(int));
for(int i=0; i<row; i++)
{
    for(int j=0; j<col; j++)
    {
        ptr[i*col+j] = i+1
    }
}
```

How to find sizeof array in C/C++ without using sizeof operator

In C lang. when we increment or decrement pointer then pointer point to next or previous memory loc of that type.

```
int data[5] = {10, 20, 30, 40, 50, 60};
```

data = array of integer

&data[i] pointer to integer and its value is addr of ith elements of array.

(47) data[i] = *(data+i)

i

*(data+i) = data[i]

(data+i) = &data[i]

i=0 (data+0) = &data[0]

data = &data[0].

Note: when we put (&) before the array name then its type change. It becomes a pointer to the array.

data = pointer to first element of array.

&data = pointer to an array of 6 elements

&data+1 = pointer to next memory block (i.e add ahead of 6 integers)

*(&data+1) = add of first element of second mem block.

int size = *(&data+1) - data;

pass an array as parameter

int arr[] = {10, 20, 30, 40, 50};

void ReadArray(int *pInt)

ReadArray(arr) or

ReadArray(int arr[])

or

int row = 15;

void ReadArray (int arr[row])

Passing 2D Array:

- When we pass a 2D array as a parameter, decays into a pointer to array, rather pointer to a pointer.
- First element of the multidimensional array is another array. When when we pass 2D array then it would be split into a pointer to another array.

(i) int data[3][3]; it will split into pointer to an array of 3 integers int (*)[3];

void ReadArray (int (*ptr)[3])

ReadArray (arr)

(ii) void ReadArray (int arr[row][col])

(iii) void ReadArray (int arr[][col])

(iv) int data[3][3]; then &data would be a ~~pointer~~ add to 2D array which has 3 rows and 3 cols.

void ReadArray (int (*ptr)[row][col])

ReadArray (&data)

Note: (i) we cannot use multiplication (*) or division operator with ptr.

(ii) When we increment or decrement ptr it will point to next memory of that type

pointer comparision is invalid when it points to 2 diff mem blocks

Note: C standard doesn't allow arithmetic opr on void ptr. But GNU C allows arithmetic, size of void as 1

Arrays

* int data[20] = {0}; // Initialize all elements to 0

Designated initializers

- allows us to initialize specific elements of the array in any seq. we don't have to initialize the array from beg.
- the size of the array is not given then the largest initialized posn become the size of the array and all uninitialized posn initialized to 0.

int arr[20] = { 1, 2, 3, [15] = 40, 5, [13] = 80, [18] = 89 };

Here a[0] = 1
a[1] = 2
a[2] = 3
a[13] = 80
a[15] = 40
a[16] = 5
a[18] = 89

and rest all 0

Ex.2 int a[6] = {[4] 29 [2] 15};

a[4] = 29
a[2] = 15

and rest all 0.

OR

int arr[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };