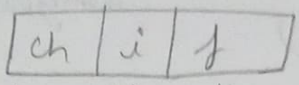
When a structure is declared no memory is allocated for the members.

If we want to access the members of a structure through structure variable, we need to use dot operator '.'

```
main()
{
  int i;
  struct one o1;
  o1.ch = 'a';
  o1.i = 10;
  o1.f = 23.5;
  pf("%c %d %f", o1.ch, o1.i, o1.f);
}
```

o1 is a variable of struct one data type

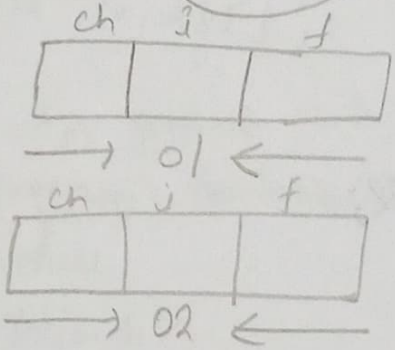


1 byte, 4 bytes, 4 bytes = 9 bytes size logically

o/p:- a 10 23.500000

Now consider,

struct one o1, o2; → 2 variables of type struct one



Logically the size of struct one is 9 bytes (1+4+4)

Rules of declaring a structure.

① Structure member can't be initialized while declaring.

② Storage type should not be mentioned for structure member

Member's storage type depends on Variable's Storage type

Structure variable initialization

eg:-
```
void main()
{
  struct one o1 = {'z', 10, 23.5};
}
```

order is must for initialization but it is not the same when assigning value individually

```
Void main()
{
  char ch;
  struct one o1={'z',10,23.5}
  pf("%c %d %f\n", o1.ch, o1.i, o1.f);
  o1.ch = 'a';
  pf("%c %d %f\n", o1.ch, o1.i, o1.f);
}
```

eg:-
```
Void main()
{
  struct one o1; // has garbage value as default
  static struct one o2; // has (zero) as default
```

eg:-
```
void main()
{
  struct one o1={'z', 10, 23.5}, o2;
  pf("%c %d %f\n", o1.ch, o1.i, o1.f);
  pf("%c %d %f\n", o2.ch, o2.i, o2.f);
  o2=o1;
  pf("%c %d %f\n", o1.ch, o1.i, o1.f);
  pf("%c %d %f\n", o2.ch, o2.i, o2.f);
}
```

o/p:-
```
z  10  23.500000
0  0   0.000000
z  10  23.500000
z  10  23.500000
```

eg:
```
Void main()
{
  struct one o1;
  printf("Enter char \n");
  scanf("%c", &o1.ch);
  printf("Enter int \n");
  scanf("%d", &o1.i);
  printf("Enter float \n");
  scanf("%f", &o1.f);
  printf("%c %d %f", o1.ch, o1.i, o1.f);
}
```

eg:
```
struct one
{
  char ch;
  int i;
  float f;
} o1 = {'a', 10, 23.5};
```

session - 8 23-09-20

#Write a program to maintain a student database using structure

```
#include<stdio.h>
void main()
{
  struct st
  {
    int rollno;
    char name[10];
    float marks;
  };
  void main()
  {
    struct st s1;
    printf("Enter rollno \n");
```

```
sf("%d",&s1.rollno);
pf("Enter name\n");
sf("%s",s1.name);
pf("Enter marks\n");
sf("%f",&s1.marks);
pf("%d %s %f\n",s1.rollno,s1.name,s1.marks);
```

}

eg:-

```
#include<stdio.h>
struct st;
{
  int rollno;
  char name[10];
  float marks;
};
void main()
{
   struct st s[5];
   int i,ele;
   ele = sizeof(s)/sizeof(s[0]);
   for(i=0;i<ele;i++)
    {
      pf("Enter roll no\n");
      sf("%d",&s[i].rollno);
      pf("Enter name\n");
      sf("%s",s[i].name);
      pf("Enter marks\n");
   }
   for(i=0;i<ele;i++)
   pf("%d %s %f\n",s[i].rollno,s[i].name,s[i].marks);
}
```

Array of structure (structure pointer)

In a structure '(->)' arrow operator means de-referencing

```
# #include<stdio.h>
  struct st
  {
    int i;
    char ch;
    float f;
  };
  void main()
  {
      struct st s1={10, "hello", 23.5}, *p;
      p=&s1;
      pf("%d %s %f\n", s1.rollno, s1.name, s1.marks);
      pf("%d %s %f\n", p->rollno, p->name, p->marks);
      p->rollno = 20;
      //p->name = "abcd"; // error
      strcpy(p->name, "abcd"); //name is changed now
      pf("%d %s %f\n", s1.rollno, s1.name, s1.marks);
  }
```

Write a program to allocate a dynamic memory for one student record, scan & print it.

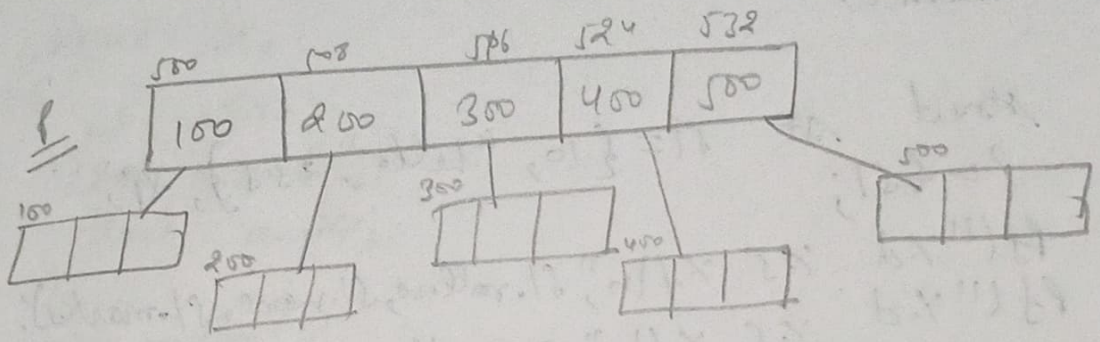
Note:- Header file will not effect the .exe file

struct is the same as prev. program.

```
Void main()
{
    struct st *p;
    p=malloc(sizeof(struct st));
    pf("Enter rollno\n");
    sf("%d", &p->rollno);
    pf("%f",
```

```
Pf("Enter name\n");
Sf("%s", p->name);
Pf("Enter marks\n");
Sf("%f", p->marks);
Pf("%d %s %f\n", p->rollno, p->name, p->marks);
}
```

write a program to allocate a dynamic memory for 5 students records. Scan it & print it.




```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct st
{
  int rollno;
  char name[10];
  float marks;
};
void main()
{
  struct st *p[5]
  int i;
for(i=0; i<5; i++)
p[i]= malloc(sizeof(struct st));
for(i=0; i<5; i++)
{
  Pf("Enter rollno\n");
  Sf("%d", &p[i]->rollno);
  Pf("Enter name\n");
```

```
        sf("%s", p[i]->name);
        pf("Enter marks\n");
        sf("%f", &p[i]->marks);
    }
    pf("----------\n");
    for(i=0; i<5; i++)
    pf("%d %s %f\n", p[i]->rollno, p[i]->name, p[i]->marks);
}
```

Allocate dynamic memory for n student records. Scan & print it.

The struct is the same as the pre. program

```
void main()
{
    struct st **p;
    int i, n;
    pf("Enter n\n");
    sf("%d\n", &n);
    p = malloc(sizeof(struct st*)*n);
    for(i=0; i<n; i++)
    p[i] = malloc(sizeof(struct st)); // allocated dynamic memory for 1 ptr.
    for(i=0; i<n; i++)
    {
        pf("Enter rollno\n");
        sf("%d", &p[i]->rollno);
        pf("Enter name\n");
        sf("%s", p[i]->name);
        pf("Enter marks\n");
        sf("%f", &p[i]->marks);
        pf("----------\n");
    for(i=0; i<n; i++)
    pf("%d %s %f\n", p[i]->rollno, p[i]->name, p[i]->marks);
}
```

Session - 81 24-09-20

In one structure, if there are variables of another structure, then it is called nested structure.

eg:
```
struct Date
{
 int date;
 int month;
 int year;
};
struct st
{
 int rollno;
 char name[20];
 float marks;
 struct Date dob;
 struct Date doj;
};
```

For a structures, a memory is allocated only when we create a variable to the structure member

In one structure, we cannot put a variable as a member of the same structure type. This is bec. the structure declaration is not yet ended.

eg:
```
struct st
{
 int rollno;
 char name[20];
 float marks;
 struct Date
  {
    int date;
    int month;
    int year;
  }
}
```

```
struct
 struct st
 {
  int rollno;
  float marks;
  struct st dob; ✗ error
 }
```

Self referential structure : In one structure, one of the member is the pointer of same structure type, then that structure is called self referential structure

→ This type of structure we need to used in data structure (linked list)

eg:-
```
struct st
{
  ===
  ===
  struct st *ptr;
};
```

```
#include<stdio.h>
struct st
{
  int rollno;
  char name[10];
  float marks;
};
void main()
{
  struct st *p;
  p=malloc(sizeof(struct st));
  pf("Enter rollno\n");
  sf("%d",&p->rollno);
  pf("Enter name\n");
  sf("%s",p->name);
  pf("Enter marks\n");
  sf("%f",&p->marks);
  pf("%d %s %f\n",p->rollno,p->name,p->marks);
}
```

How to handle, if one of the structure member is a pointer?

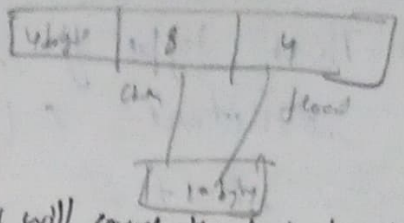
Another ex

```
struct st
{
  int rollno;
  char *name;
  float marks;
};
```

If the structure member is a pointer to hold the add. of the names, we have to allocate a dynamic memory to it. If no memory is allocated, it gives some unexpected result.

```
void main()
{
struct st *p;
p=malloc(sizeof(struct st));
p->name=malloc(10);
Pf("Enter name,rollno,mark\n");
sf("%d %s %f",&p->rollno,&p->name,&p->mark);
}
```

free(p);
p=0



This will cause the 10 bytes of memory leak
It has to be freed in reverse order
free(p->name);
free(p);

Session - 8

25-9-20

Bit field :- It is the method of allocating memory in the form of bits.

This mechanism is used in IPV4 header, CAN protocol, TCPIP, TLP header etc.

Bit field Rules :-

① Bit field is used in structure and union

Syntax:

```
datatype variablename : bits;
```

eg:- int a : 3;

3 bits are allocated for the variable 'a'.

② Declare unsigned bitfield so that all the bits acts as data bits.

③ Float/double bit field is not allowed.

eg:-
```
#include<stdio.h>
struct one
{
char ch;
int m:3;
int i;
float f;
};
```

In this case the last bit is ~~data~~ sign bit

⇒ If this is declared unsigned all bits are data bits.

```
void main()
{
struct one o1;
o1.m = 7;
printf("%d\n", o1.m);
}
```

eg:- unsigned int m : 3;
gives
o/p = 7

④ & (address operator) can't be applied.
⇒ This can't be scanned. Address are allocated in the form of bytes.

⑤ sizeof() operator can't be applied to bit fields

⑥ There is no seperate format specifier to deal with this bit field variable.

eg: if unsigned int m : 3;
⇒ o1.m = 9; ← stack overflow happens bec, 9 needs 4 bits

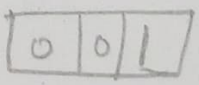
How to find the size of the bit field variable?

```
eg:- void main()
{
struct one o1;
int c=0;
o1.m = 1; ← Initialization;
while(o1.m)
{
c++;
o1.m = o1.m<<1;
}
printf("c = %d Bits\n", c);
}
```

```
struct one
{
unsigned int m:3;
int i;
};
```

m is a variable of size 3 bits

m is a variable of size 8 bits holding a value 1



o/p
c = 3 bits

Passing structure variable/members to a function

```
eg:- #include <stdio.h>
struct A
{
int i;
```

```
    char ch;
    float f;
};
Void print(char, int, float);
Void main()
{
 struct A a1 = {'a', 10, 23.5};
 Print(a1.ch, a1.i, a1.f);
}
void print(char ch, int i, float f)
{
 Printf("In print %c %d %f\n", ch, i, f);
}
```

#Design a function which should return int, char & float. To return multiple types, a structure can be used.

```
struct A
 {
  char ch;
  int i;
  float f;
};
Void A return_struct(Void);
Void main()
{
 struct A a1;
 a1 = return_struct(void);
 Printf("%c %d %f\n", a1.ch, a1.i, a1.f);
}
struct A return_struct(Void)
{
 struct A a2 = {'a', 20, 23.5};
  return a2;
}
```

Here a2 is returned and a1 catches this (a2)

This is call by value method

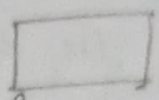
After a2 is copied to a1 this stack frame gets deleted

Session - 8.3 26-09-20

Design a function to allocate dynamic memory for one Student record & return the dynamically allocated memory address.

```
#include <stdio.h>
#include <stdlib.h>
struct st
{
  int rollno;
  char *name;
  float marks;
};
void main()
{
  struct *p;
  p = my_alloc_struct();
  Printf("Enter rollno \n");
  Scanf("%d", &p->rollno);
  Pf("Enter name \n");
  Sf("%s", p->name);
  Pf("Enter marks \n");
  Sf("%f", p->marks);
  Pf("%d %s %f \n", p->rollno, p->name, p->marks);
}
struct st *my_alloc_struct(void)
{
  struct st *q;
    q = malloc(sizeof(struct st));
    q->name = malloc(10); // if name is an array, this statement is invalid.
  return q; // address of q should not be returned because this is an auto variable.
}
```

#Design a function to allocated dynamic memory for one student record but the fun. return type should be void.



we need to catch this with double pointer

we are passing address of a pointer to the fun. of a catching it with the double pointer.

```
#include<stdio.h>
#include<stdlib.h>
struct st
{
  int rollno;
  char *name;
  float marks;
};
void my_alloc_struct_1(struct st **);
void main()
{
 struct st *p;
 my_alloc_struct_1(&p);
```

↳ passing the address of the pointer

```
 Pf("Enter rollno\n");
 Sf("%d",&p->rollno);
 Pf("Enter name\n");
 Sf("%s",p->name);
 Pf("Enter marks\n");
 Sf("%f",p->marks);
 Pf("%d %s %f\n",p->rollno,p->name,p->marks);
}
void my_alloc_struct_1(struct st **q)
{
  *q=malloc(sizeof(struct st));
  (*q)->name=malloc(10);
}
```

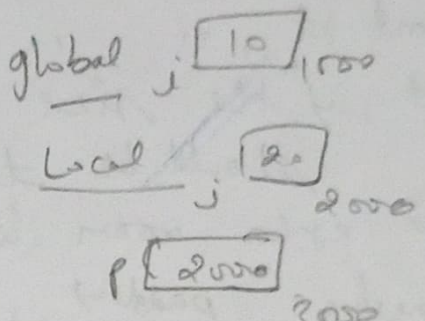
Eg:-

```
int i=10;
main()
{ int i=20;
```

```
    int *p = &i;
    abc(p);
    Pf("%d\n", *p); //20
}
void abc(int *q)
{
    q = &i;
}
```

Another situation

```
int i=10;
void main()
{
    int i=20;
    int *p=&i;
    abc(&p);
    Pf("%d\n", *p); //10
}
void abc(int **q)
{
    *q = &i;
}
```



3000 is passed in fun.
*q = &i; → we make the pointer to point to global.

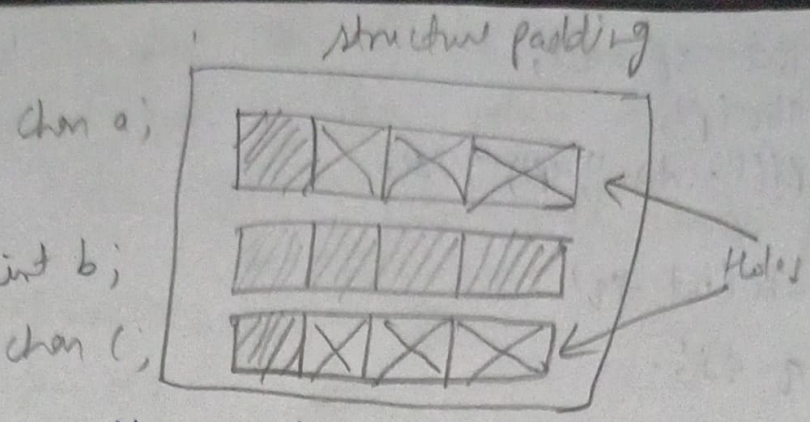
Structure padding & holes

program
compiler
OS
processor

8-bit controller → process 1 byte at a time

→ Structure padding is a compiler dependent process. Main intention of structure padding is to increase the speed of processing.

The wastage of memory is called as holes & adding of extra byte in a structure is called a structure padding.



For char, it should be multiple of 1
For short it, " " " ; 2
For int, if the size is 4 or more than 4, it should be the multiple of 4.
This extra room is created due to the rules.

Structure padding

advantage
↓
Faster processing

Disadvantage
↓
memory gets wasted.

Session - 84

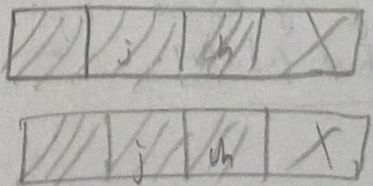
28-9-20

Bytes are fetched in the form of 4 bytes. To avoid structure padding, all similar data types should be put together.
= Order plays a major role.

eg: struct A
{ short int i;
char ch;
} a1;
→ Now, sizeof(a1) gives 4 bytes

Consider struct A
{ short int i;
char ch;
short int j;
char ch1;
} a1;

Here sizeof(a1) → 8 bytes



To avoid structure padding, use #pragma pack(1)

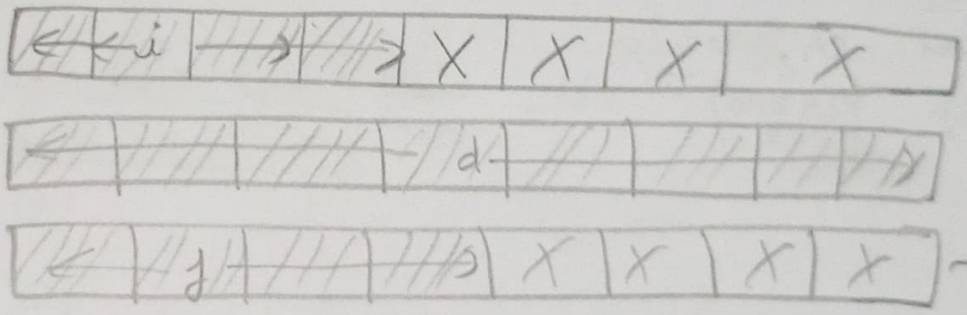
eg:
```
#include<stdio.h>
#pragma pack(1)
struct A
{
   int i;
   double d;
   float f;
}a1;
int main()
{
   Pf("%ld\n", sizeof(a1));
}
```

o/p 16 bytes

Without pragma pack, the o/p is 24

The largest datatype size's multiple is considered. Here, the data is packed in a pack of 8.

⇒



When pragma pack(1) is given, the speed of processing is reduced

```
Struct A
{
    int i;
    double d;
    float f;
}a1;
```

size of(a1) → gives 24 bytes

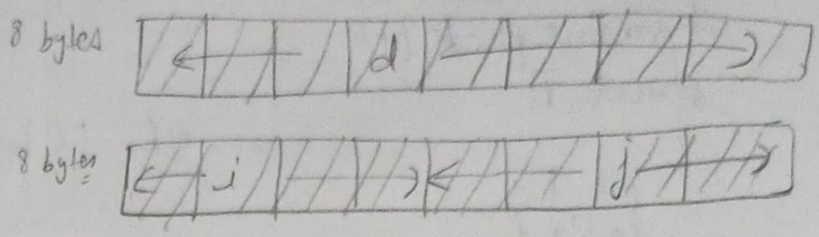
#pragma pack(1) → no structure padding

any variable is the multiple of 1 byte.

```
Struct A
{
   double d;
   int i;
```

float f;
} a1;

sizeof(a1) = 16 bytes
This is due to the order of member declaration.



Eg:

```
struct A
{
  int i[5];
  double d;
  float f;
}a1;
```

bytes are packed in 8 bytes pack

sizeof(a1) = 40 bytes
8 bytes pack
i[5]

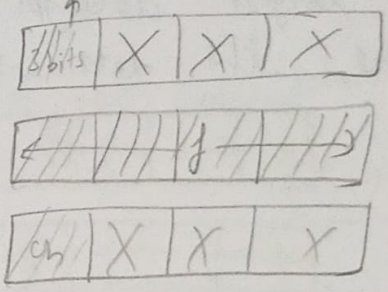
Eg:

```
struct A
{
  int i:3;
  float f;
  char ch;
}a1;
```

3 bits

sizeof(a1); → 12 bytes

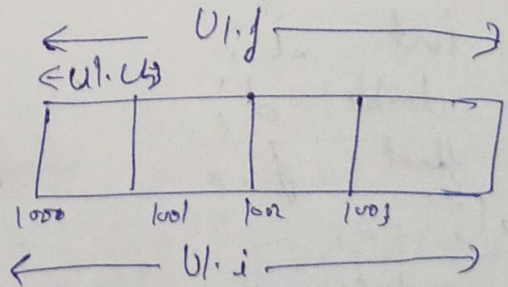
Largest datatype = 4 → 4 byte package



Union:-

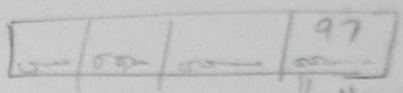
Union is one of the user defined datatype, which is a collection of different types of data which are in same memory location.

```
Union U
{
  char ch;
  int i;
  float f;
}u1;
```



Note:- In Union we can process only one member at a time.

eg:
```
Union U
{
  char ch;
  int i;
  float f;
} U1;
Void main()
{
  U1.i = 258;
  Pf("%d %d\n", U1.ch, U1.i); // 2  258
  U1.ch = 'a';
  Pf("%d %d\n", U1.ch, U1.i); // 97  353
}
```



⇒ Union will hold the latest value stored.

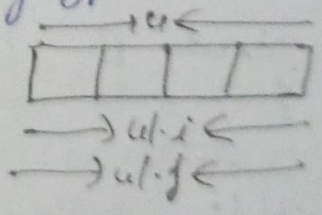
24 Session - 85

29-9-20

→ Size of the union is the ^size of largest data type.

```
#include<stdio.h>
union U
{
  int i;
  float f;
};
Void main()
{
  union U U1;
}
```

← U1 is a variable of type union U



A union can process only one data at a time, so that it can save memory.

```
#include<stdio.h>
union U
{ int i;
  float f;
}; void main()
{ union U u1;
```

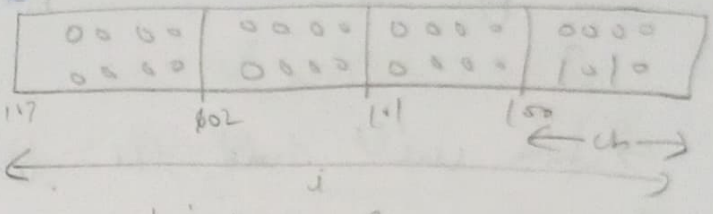
```
    int pos;
    u1.j = 23.5;
    for(pos=31; pos>=0; pos--)
    pf("%d", u1.i >> pos & 1);
    pf("\n");
}
```

Using union find the endianess of the processor

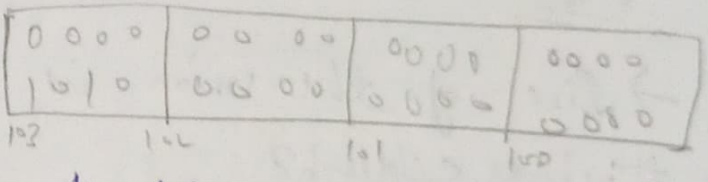
```
#include<stdio.h>
union u
{
  char ch;
  int i;
};
void main()
{
  union u u1;
  u1.i=10;
  if(u1.ch==10)
  pf("Little \n");
  else
  pf("Big \n");
}
```

→ compare with whatever value is stored in the first byte from the lower address

u1.i = 10 ⇒ Little endian



u1.i = 10 ⇒ Big endian



Any value other than '0' & -1 can work.

If '0' or -1 are stored in i instead of 10, we can't know/show the endiness, bec. in 0 all bits are zero '0' & in -1 all are (32) bits 1's

```
Union A
{
  struct B
  {
    ==== 4
    ==== 8
  }bl;<size 16 bytes
  struct C
  {
    ==== 8
    ==== 4
  }cl;<12 bytes
}al;
```

The size of Union A.al is 16 bytes

Union considers the largest size of its members.

Here, struct B has the largest size (i.e) 16 bytes.

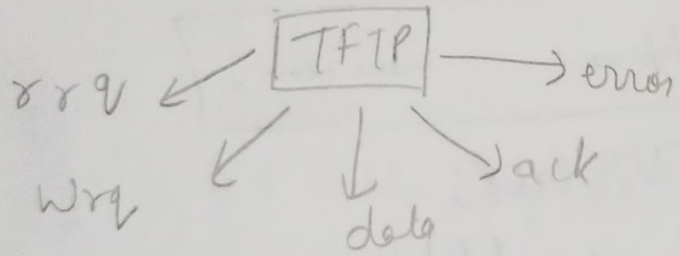
eg: struct A
{
union B
{
4 bytes
} b1; → 4 bytes
union C
{
8 bytes
} c1; → 8 bytes
} a1;
1 of 2 = 3 bytes

union B.b1 & C.c1 will hold the size of its largest struct mem. But struct A.a1, will check 2 diff. locations for its members union B & C.

Usage of a union

eg:- TFTP → Trivial File Transfer Protocol

TFTP has 5 msgs to send, but only one msg can be sent at a time.



To process one msg at a time, we pack of all these 5 msgs in a union.

eg: union - TFTP
{
struct rrq
{
} r1;
struct wrq
{
} w1;
struct data
{
} d1;
struct ack
{
} a1;
struct error
{
} e1;
}

Here, at a time, only one msg can be processed/ travelled.

To ensure that, at a time only one packet is travelled in a network, we can use union.

Session - 8

30-9-20

Userdefined datatype → typedef
→ enum

Typedef :-

→ Typedef is userdefined datatype
→ Typedef does not create a new datatype but we can create another name to a datatype

= How to design typedef :-

→ int
→ structure
→ newname datatype
→ array
→ function
→ pointer

⇒ Header files contain function declaration

Syntax:

```
typedef oldatatypename new_name;
```

eg:- typedef int INTEGER;
without typedef, int is datatype
If typdef is used INTEGER is treated as another name to int datatype.

```
#include<stdio.h>
typedef int INTEGER;
void main()
{
int i;
INTEGER j;
Pf("%d%d",sizeof(i),sizeof(j));
}
```

o/p
4 4

We can typedef more than one variable

eg
```
#include<stdio.h>
typedef int INTEGER, I, INT;
void main()
  int i;
  INTEGER j;
  I k;
  INT l;
  pf("%d %d %d %d", sizeof(i), sizeof(j), sizeof(k), sizeof(l));
}
```
o/p
4 4 4 4

=Typedef uses:-

① ↑ Readability
② ↓ complex declaration
③ Duplicate name will not allocate memory.

eg
```
#include<stdio.h>
typedef unsigned char UCHAR;
void main()
{
  UCHAR u1;
  pf("%d", sizeof(u1));
}
```
o/p
1

Array typedef

```
#include<stdio.h>
typedef int arr[5];
void main()
{ arr a1, b[3]
  pf("%d\n", sizeof(a1));
}
```

b is an array of 3 arr's.
It is a 2d array
⇒ b[3][5]
b[5][5] ✗ wrong leading
✗

typedef is used to create a user-defined datatype.

Pointer typedef:-

```
#include<stdio.h>
typedef int *ptr;
void main()
{
  int i=10;
  iptr p;
  Pf("%d\n",*p);
}
```

iptr → integer pointer data type.

o/p: 10

storage type cannot be used in typedef

```
#include<stdio.h>
typedef int *ptr;
void main()
{
  int i=10;
  static iptr p, *q;
  p=&i;
  Pf("%d\n", *p);
  q=&p;
  Pf("%d\n", **q);
}
```

o/p: 10, 10

*p → ptr; **q → double ptr

Typedef can be declared either globally or locally

typedefing a structure:-

Method-I

```
#include<stdio.h>
typedef struct one
{
  char ch;
  int i;
  float f;
} O;
void main
{
  O o1={'z',10,23.5};
```

O → another name for structure; without typedef 'O' will be considered as the variable for struct one.

```
printf("%c %d %f\n", o1.ch, o1.i, o1.f);
}
```

Method II

1 → Finishing the structure declaration
2 → typedefing it.

```
#include<stdio.h>
struct One
{
  char ch;
  int i;
  float f;
};
typedef struct one O;  ← this typedef can be given inside main() also
void main()
{
  O o1={'z', 10, 23.5};
}
struct one
{
  char ch;
  int i;
  float f;
};
typedef struct one O;
typedef O o1;
void main()
{
  o1 o1={'z', 10, 23.5};
}
```

ENUM :-

→ using enum, we can provide a name to constants.

eg:- enum colour {RED, GREEN, BLUE};
0 1 2

→ Using enum we don't create a data type, we just give names to constants using enum.