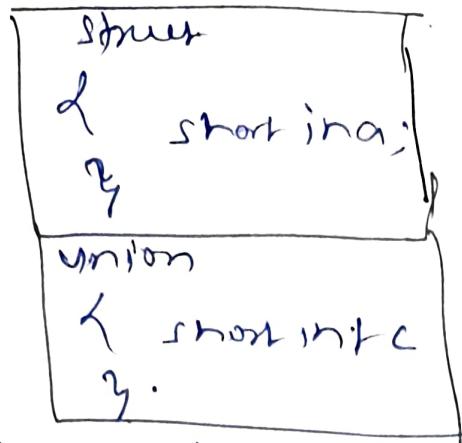


(26)



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

Ex: struct myData

```
    { union  
        { short int a;  
         short int b;  
         };  
        short int c;  
    };
```

struct myData md;

md.a = 10 \checkmark
md.b = 20 \checkmark valid
md.c = 30

Ex: 2
~~struct~~ myData

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

?;

shows my Data.m:

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 C99

enable users 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 at least 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 or info. about its members.

Incomplete type:

array type where dimension not specified

or struct or union where members not specified.

ex: struct stud

{ int roll;

}; ~~char name[5]; // incomplete array~~
as last member

How to use flexible array

~~for~~ int m=5;

struct stud *ptr = malloc (sizeof(stud)
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[5]; // ↗
size will be
8 bytes we
should ignore

};

flexible array member

struct stud s1

while calculating size of

struct.

Why does a flexible array require
dynamic memory

(29) struct hawk in C

~~↳ this technique permits user to create a variable length number 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 hole.~~

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 implemented from defined

polymorphism in C with the help of function pointer

~~↳ polymorphism with some function pointer~~

represents generic ~~for~~ containers and the function pointer behaves like C++ member function

ex:

typedef struct

d.

int (*opensocket) (void)

int (*closesocket) (void)

int (*readfromserver) (int, char*, int)

int (*writetoserver) (int, char*, string)

} sCommStruct;

→ generic combiner for server commFor tcp/ip: initialize function pointer with
tcp/ip related api and for udp we
should initialize them with ~~the~~ udp apiHow to use above structsCommStruct *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 a, size_t n);

✓ ptr = starting add of mem to be filled

a = value to be filled

✓ n = no of bytes to be filled starting from ptr to be filled.

char name[50] = "Rajkalash_Raybler";

memset(name, '0', 8)

O/p: Rajkalash.0000000"Raybler

✓ 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: und predictable: since memset works character by character.~~

Structure padding in C

(32)

* ~~Memory alignment affects the performance of the copy. And the processor takes some extra steps to access the unaligned mem.~~

→ alignment is the micro optimization technique.

* 32 bit processor, word size 4

Note: Generally compiler handles the scenario 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) before the member of the structure or union for alignment.

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

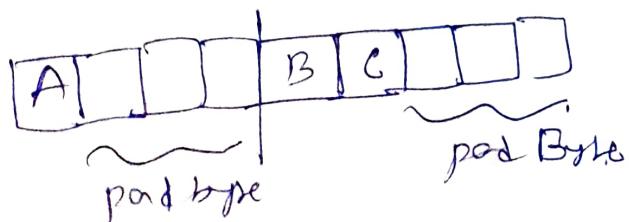
Ex struct stud

h. char A

int B

char C

y;



Total 6 bytes padding.

(33) struct stud

{ int A;
char B;
char C;

?;



2 byte padding

OR

{ char A;
char B;
int C;

?;



2 byte of
padding

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

How to avoid structure padding

① By using `#pragma pack(1)`

② attribute (`--attribute--`)
`--))`.

ex: ① #pragma pack(push,1)
typedef struct

{ double A
char B
char C

? std;

#pragma pack(pop)

It tells that the alignment is 1 byte
so no padding

ex: ~~#pragma pack(push, 2)~~
 ↓
 Struct Stud
 ↑
 alignment is 2 byte

(34)

↳ double A;
 char B;
 char C;
 {
 ~~#pragma pack(pop)~~
 }

ex-3

~~#~~

~~#pragma pack(push, 4)~~

typedef struct

alignment 4 byte

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

ex-4 ⑩ Default alignment

typedef struct

↳ double A;

alignment = 8 bytes

char B;

& 6 bytes for

char C;

: padding.

} Stud;

How to find size of structure without using
sizeof operator?

ex: typedef struct

↳ double A;

char B;

 } Stud;

(35)

↳ Student:

Student myInfo[2] = {0}.

student *ptr=NULL

$$\text{int sizeOfStruct} = \underline{(\text{char}^*)(\text{ptr} + 1)} - (\text{char}^*)\text{ptr}$$

OR

ptr++ will also give size if
increases are block by ptr by size of block.

what struct have

→ used to create flexible length member
by creating a zero size array inside
structure. → to dynamically manage
mem.

why not use pointer instead of struct have
→ extraspaces for pointers + 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[20] ~~while calculating~~}; ~~the size of string~~

have we should ignore

or the array

pointer inside structure

(36)

typedef struct

{ int i;

char *pcName;

} Info;

Info myInfo = { 20, "Ray" };

myInfo.pcName

pointer to structure:

typedef struct

{ int *pI; /*
char *pcName;

} Info;

[Info * ptrInfo = NULL;]

ptrInfo = malloc(sizeof(Info))

ptrInfo->pI = malloc(sizeof(int))

ptrInfo->pcName = malloc(sizeof(~~char~~ or
char))

~~char~~ or
char)

37)

Array of structure

struct Employee

 { preference of (.) and ([]) does
 is same and amoutati-
 vity is from left to
 right }

 int eid;
 int salary;

 char name[10];

}

struct Employee e[10];

e[0].eid = 1001

e[0].salary = 20000

strcpy(e[0].name, "Rajkumar");

Union — main purpose is to save mem.

avoids ~~overwriting~~ ~~overlapping~~ ~~overlapping mem.~~ Fused to store different data types in the same mem loc which means an efficient way of using mem but not at the same time.

* At union declaration, union's tag is Optional and each member of union defines as separate

union myData

{
 int Age;
 float fees;
 char name[10];

} data // optional

→ union variable declaration

union myData

{
 int Age;
 float fees;
 char name[10];

} union myData m, Age; Age

(*) up) • Age
or

→

up → Age

37)

Array of structure

struct Employee

{ int eid;

int salary;

char name[10];

}

~~The precedence of [] and () do
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, "Rajkumar");

Union - main purpose is to save mem.

~~avoids~~ Union - used to store different data types in the same memory. It is some mem loc which means an overlapping loc. An efficient way of using the mem but not at the same time.

~~* At~~ union declaration, union tag is Optional and each member of union defines as memory

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, *up;

(* up) . Age
or

~~m1~~ Up . Age

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

Union Test

↳ float fAmount;

int iAmount;

}

union Test dt;

dt.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

{ }

→ get permission

{ }

* Important points:

* we can create bit field using union.

union Data

{ }

unsigned int a:4

unsigned int b:4

int c

{ }

(3) * 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.

with union Data

h int a;

};

Union Data d1 = { .a = u } ; ←

* flexible arrays is not supported in union.

Bitfield

The order of autoconversion of bit fields is implementation defined.

struct Bitfield

h int data1: 1

int data2: 2

int data3: 3

};

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

⑧ struct Bitfield data

data.data1 → illegal

⑪ we cannot create an array of bitfield.

struct Data

h int a: 1

int b: 2

?;

int c[5]: 5: → illegal

④ (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 unspecified.

↙ The bit field declaration 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 only was placed.

Ex: typedef struct

{ unsigned int dater:5;

 unsigned int dater:8;

} dater

~~size of~~

for Dater d1

sizeof(d1)= 4

typedef struct

{ unsigned int a:5

 unsigned int:0; // Unnamed Bit Field

 unsigned int c:8

} UnnamedBitField

④

Unnamed Bitfield unbf:

size of (unbf) = 8

⑤ we can not calculate the size of bitfield in using sizeof operator

~~struct bit~~

typedef struct

h 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 * p1

void * p2

What's the size of void pointer?

its size is same as character pointer.

Behavior Deferring void pointer

(42)

For void pointer we cannot use (*) indirectly operator directly. before using * to void phr 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?

This is used for code reusability.
 To ~~store the add of any object~~ whenever required we can get back the object through the indirection operator through proper casting.

Arithmatic operation void pointer

int iparam[] = {100, 200, 300}

void *pvData

pvData = &iparam[0];

(pvData)+ incorrect

(int*)(pvData)+ correct ✓

~~(Q3)~~ ~~Q3~~: A T. C. standard. ~~an~~ means opn 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.~~ On void pointers to arrays
the size of void is L.

Ex: $\text{int } \text{spData} = \{100, 200, 300\}$
 $\text{void } * \text{pvData} = \&\text{spData[0]}$
 $\text{pvData} = \text{pvData} + 1$
Op: $(\text{int } *) \text{pvData}$
Op: 300 or compilation error

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

NULL pointer

A-T-C standard on integer written exp with
the value 0 or such an exp. cast to type
void * is called null ptr constant. If a
number constant is converted to pointer type
then the resulting pointer is called null ptr
Define NULL: $(\text{void } *) 0$

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 \rightarrow prevents the ptr to become
pointer A-T-L or a also new null pr constant.

Dangling pointer

so we use sizeof Operator on NULL pr,
you. The results similar to other pointers
which means if pointer size of platform is
byte. sizeof Operator on NULL gives off
4 bytes.

it is $\ast \text{ptr} = 0$ $\text{int } \ast \text{ptr} = 0$

wild pointers

~~initialized pointers are known as~~. The behaviour of wild pointers is undefined since they point to some arbitrary mem loc.

$\text{int } \ast \text{ptr}$; wild pointer

Dangling pointers!

when referencing objects deleted or deallocated
come changing the value of pointers

- (1) when pointer variable goes out of scope
- (2) After destroying the stack frame,
- (3), deleting from memory explicitly

e.g. $\text{int } \ast \text{ptr} = (\text{int } \ast)$ malloc (sizeof \ast
 sizeof(int))

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

ptr=NULL pr is no more dangling
pr.

(iii) Difference b/w $\star \& p$, $\star p++$ and $\star ++p$

- (i) precedence of postfix (++) higher than prefix \star
- (ii) associativity of \star " is left to right
- (iii) " prefix (++) right to left
- (iv) precedence of prefix (++) and \star same and left to right associativity
- (v) precedence of postfix (++) is higher than \star and associativity is also diff

$$\left. \begin{array}{c} \text{precedence}(\text{postfix }++) > \text{prefix }++ \\ > \star \end{array} \right\}$$

$\boxed{(\text{preced}) \text{ of } \text{prefix} = (\text{preced}) \text{ of } \star}$

(i) Ex: $\text{int arr} = \{100, 200, 300\}$
 $\text{int } *p = \text{arr}$
 $\underline{\underline{++}} \star p \approx \underline{\underline{++}}(*p) \quad 0 | p = \underline{\underline{100}}$

$$\underline{\underline{*p++}} \approx \star(p++) \quad 0 | p = \underline{\underline{200}}$$

$$\star +p \approx \star(+p) \quad 0 | p = \underline{\underline{300}}$$

Dynamically allocate 1D and 2D array

$\text{int } *p\text{Buffer} = \text{malloc}(\text{7} * \text{sizeof}(\text{int}))$

$\text{int } *p\text{ptr} = (\text{int } *) \text{malloc}(\text{7} * \text{sizeof}(\text{int}))$

2D array array

~~By using pointer to pointer~~

$\boxed{\text{int } *p\text{arr} = \text{NULL}}$

$x\text{ptr} = (\text{int } **) \text{malloc}(\text{7} * \text{sizeof}(\text{int}))$

(46)

for (int i=0; i<r; i++)

ptr[i] = (int*) malloc(sizeof(int));

3×5

using single pointerint *ptr = NULL;

int row, col;

ptr = (int*) malloc(sizeof(int) * row * col);

for (int i=0; i<row; i++)

for (int j=0; j<col; j++)

~~ptr + i * col + j~~

ptr[i * col + j] = j + 1

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

* In C long, when we increment or decrement pointers from the pointer point to next or previous memory of that type.

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

data = array of integer

&data[i] pointer to integer and its valueis address of ith elements of array

Q7

$\text{data}[i] = *(\text{data} + i)$

~~i~~ \rightarrow

$*(\text{data} + i) = \text{data}[i]$

$(\text{data} + i) = \& \text{data}[i]$

$i = 0$ $(\text{data} + 0) = \& \text{data}[0]$

$\text{data} = \& \text{data}[0]$

Note: when we put $(\&)$ before the array name
there is type change. It becomes a pointer to
the array.

$\text{data} =$ pointer to first element of array.

$\& \text{data} =$ pointer to an array of 6
elements.

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

~~$\& (\text{data} + 1)$~~ = add of first element
of second mem block.

$\boxed{\text{int size} = *(\& \text{data} + 1) - \text{data};}$

pass an array as parameter

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

void ReadArray(int *pInt)

{

 ReadArray(arr) or

 ReadArray(int arr[])

ex
int arr = 15;

void ReadArray (int arr[rows])

Passing 2D Array

- ✓ When we pass a 2D array as a parameter, it decays into a pointer to array, which is a 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.

① int data[3][3]; if we split into pointer to an array of 3 integers int (*)[3].

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

ReadArray (arr)

② void ReadArray (int arr[rows][cols])

③ void ReadArray (int arr[3][3])

④ int data[3][3]; here ~~arr~~ &data would be a ~~addr~~ pointer to 2D array which has 3 rows and 3 cols.

void ReadArray (int **ptr[rows][cols])

ReadArray (&data)

Note: ① We cannot use multiplication (*) or division operator with ptr.

② When we increment or decrement ptr, it will point to next memory of that type.

~~* pointer composition is invalid after 12 points~~
~~for 2 diff mem. blocks~~

~~C standard does not allow arithmetic op on void ptr. But UNIX C allows arithmetic size of void on L~~

Arrays

* $\text{int data[20]=}\{0\} / \text{Initialize all elements to 0}$

Designated initializers

~~It allows us to initialize specific elements of the array in any seq. we do not have to initialize the array from beginning.~~
~~If 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.~~

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

Here $a[0]=1$

$a[1]=2$

$a[2]=3$

and so on?

$a[13]=80$

$a[15]=40$

$a[16]=5$

$a[18]=89$

Ex. 2 $a[6]=\min([4]=29, [12]=15);$

$a[4]=29\}$ and rest is all 0.

$a[2]=15$

With all OR

$\text{int arr[100]=}\{(0-9)=1, [10-99]=2, [100]=3\};$