Advanced C concepts:

Endianess: How the data will be stored in a computer memory is called as Endianess.

These are of two types:

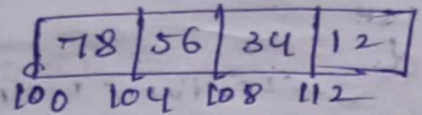
1) Little Endian. → LSB is in lower memory address.
2) Big Endian. → MSB is in lower memory address.

Example: int num = 0x12345678.

MSB LSB

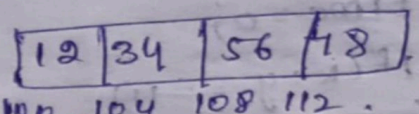
Little Endian

num



Big Endian.

num



* To check the wether the system is little Endian or Big Endian:

```
#include<stdio.h>
int main()
{
    int num=0x12345678;
    char *ptr=(char*)&num;
    if(*ptr==0x78)
    {
        printf("Little Endian");
    }
    else
    {
        printf("Big Endian");
    }
}
```

O/p: Little Endian.

Sizeof operator:

⇒ the size of the pointer is depends on the architecture:

→ If it is a 64 bit architecture, the size of the pointer is 8 bytes

→ If it is a 32 bit system, the size of the pointer is 4 bytes.

⇒ Between the two pointers, we can perform only addition and subtraction.

⇒ If only one pointer & one constant is there, we can perform only subtraction and addition.

* Pointers: Pointer stores the address of the variable. Pointer stores the address. This address should be always an integer.

Pointer arithmetic: If we perform the arithmetic operations on Pointers. It is called as pointer arithmetic.

int * represents integer Pointer.
char * represents character Pointer.

& → to get the address from the Pointer.
* → go to the particular address and fetch the value

referencing.
Dereferencing.

Rules:
1) Pointer is an integer.
2) Referencing & Dereferencing.
3) Pointer means containing
4) Endianess.
5) Pointer arithmetic.
6) Void pointer & Null pointer (pointing to nothing).
7) Static Vs Dynamic memory allocations.

Why Pointers:
1) It is used to reduce the code and improve the performance
2) To increase program execution speed.
3) to return multiple values.
4) Used in dynamic memory allocations like calloc, malloc etc.
5) For accessing array elements.

Example:
```
#include<stdio.h>
int main()
{
    int num = 5;
    int *ptr;
    ptr = &num;
    printf("%d", *ptr); // 5
}
```

num [5] 100
ptr = [100]

*100 ⇒ 5

data = *ptr
ptr = address
*ptr = value

Pointer arithmetic example:
```
#include<stdio.h>
int main()
{
    int arr[3] = {1,2,3};
    int *ptr = &arr[0];
    for(int i=0; i<3; i++)
    {
        printf("%d", *(ptr+i));
    }
    return 0;
}
```

O/P: 1 2 3.

0	1	2
1	2	3
100	104	108

1) *(ptr+0) ⇒ *100 ⇒ 1
2) *(ptr+1*4) ⇒ *(100+4) ⇒ *104 ⇒ 2
3) *(ptr+2*4) ⇒ *(100+8) ⇒ *(108) ⇒ 3

Why pointers:

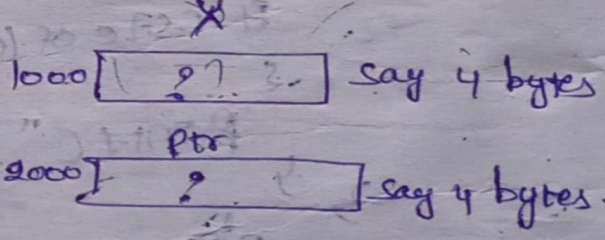
1) Returning more than one value from a function.
2) To achieve the similar results as of "Pass by value".
3) Parameter passing mechanism in function, by Passing the reference.
4) To have the dynamic allocation Mechanism.

Rule 1: Pointer is an integer:

→ A pointer is always an integer. Because, the address of the pointer is always integer only.

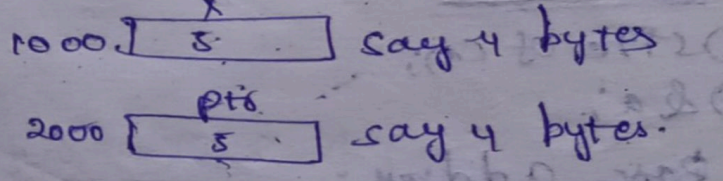
Example:

```
#Include<stdio.h>
Int maine)
{
    Int x;
    Int *Ptr;
    x=5;
    Ptr=5;
    return 0;
}
```



Example:

```
#Include<stdio.h>
Int maine)
{
    Int x;
    Int *ptr;
    x=5;
    ptr=5;
}
return 0;
```



• So, pointer is an integer.. But they may not be of same size.

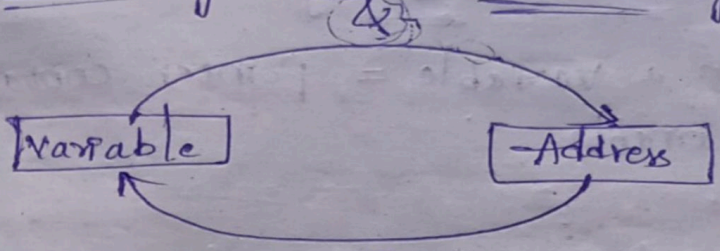
32 bit system = 4 Bytes.

64 bit system = 8 Bytes.

```
main()
{ int x=5;
  int *ptr=&x;
  pf("%d", *ptr);
}
```

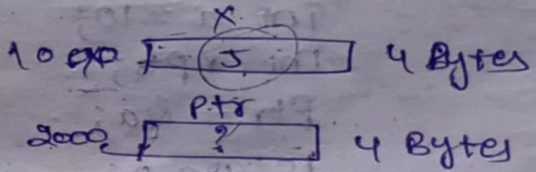
O/P: 5

Rule 2: Referencing and Dereferencing:



Example:

```
#Include<stdio.h>
int main()
{
    int x;
    int *ptr;
    x=5;
    return 0;
}
```



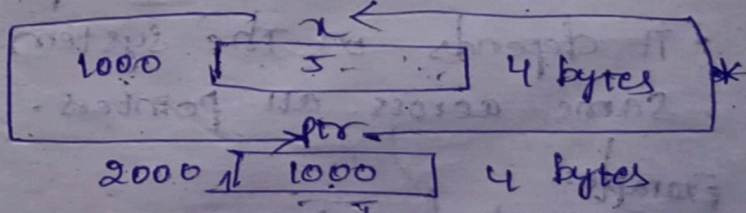
→ Goto the location 1000 and fetch its value:

So, *1000 → 5'

Example:

```
#Include<stdio.h>
int main()
{
    int x;
    int *ptr;
    x=5;
    ptr=&x;
    return 0;
}
```

&



- a pointer should contain the address of a variable.
- It should be a valid address.

→ "address of operator" (&) with variable (x) to get its address and store in the pointer.

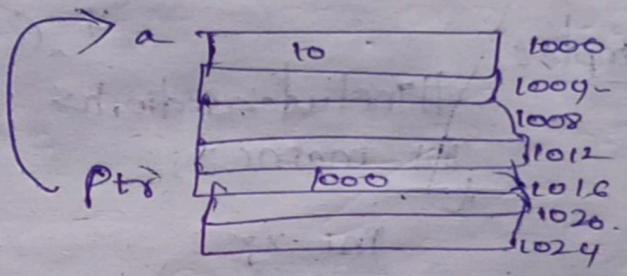
→ "indirection operator" (*) with pointer to get the value of variable (x) it is pointing to.

Rule 3: pointing means containing:

→ Pointer pointing to a variable = pointer contains the address of the variable.

Example:

```
#include<stdio.h>
int main()
{
    int a = 10;
    int *ptr;
    ptr = &a;
    return 0;
}
```



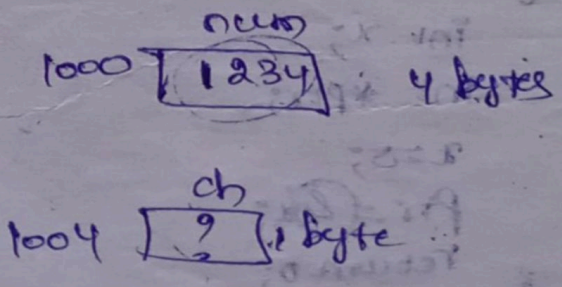
Rule 4: Pointer type:

→ Does address has a type: No.

- Address size does not depend on type of the variable.
- It depends on the system we use and remains same across all pointers.

Example:

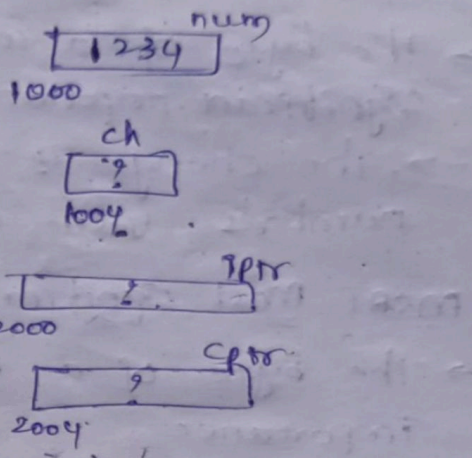
```
#include<stdio.h>
int main()
{
    int num = 1234;
    char ch;
    return 0;
}
```



From the above example, &num → 4 bytes and &ch → 1 byte.

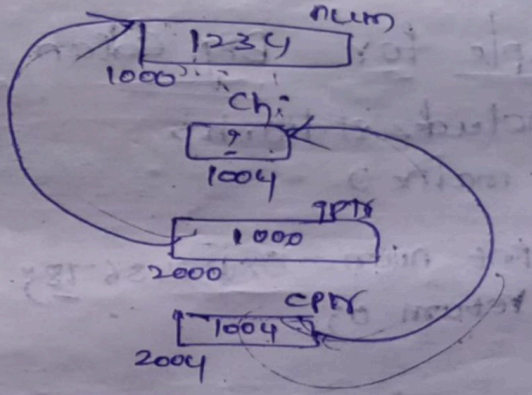
Example: we have integer type and character type.

```
#include<stdio.h>
int main()
{
    int num = 1234;
    char ch;
    int *iptr;
    char *cptr;
    return 0;
}
```



Example:

```
#include<stdio.h>
int main()
{
    int num=1234;
    char ch;
    int *iptr = &num;
    char *cptr = &ch;
    return 0;
}
```



→ with just the address, we cannot know what data is stored.

from the above example,

*cptr fetches a single byte and *iptr fetches 4 consecutive bytes.

so, the conclusion is,

(type *) → fetch size of(type) bytes.

* LSB (Least Significant Byte):

→ The byte of a multibyte number with the least significant importance.

→ The change in it would have least effect on number's value change. Ex: 00000001 = 1.

* MSB (Most Significant Byte):

→ The byte of a multibyte number with the most importance.

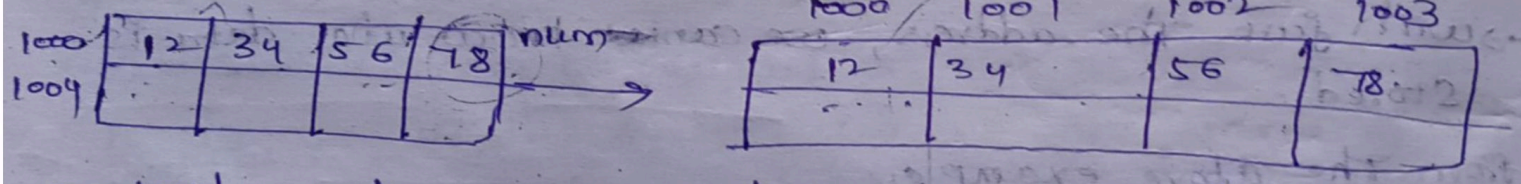
→ The change in it would have more effect on number's value change. Ex: 1000 0000 = 128.

Example for Big Endian and little Endian:

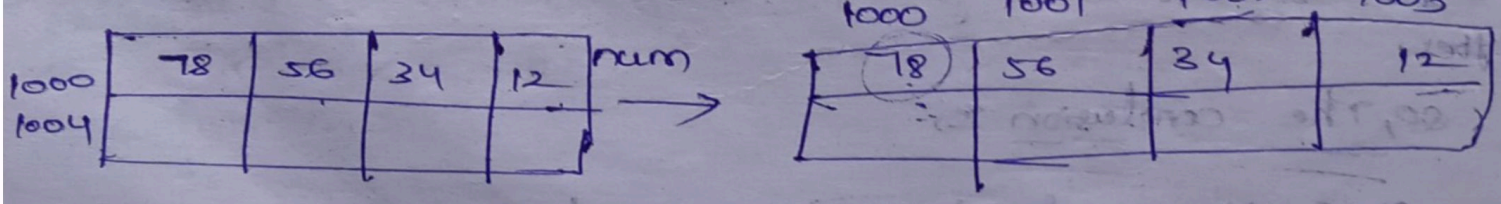
ordering of the data in the system is called endianess.

```
#include<stdio.h>
int main()
{
    int num = 0x12345678;
    return 0;
}
```

for Big Endian, → MSB in Lower memory address.

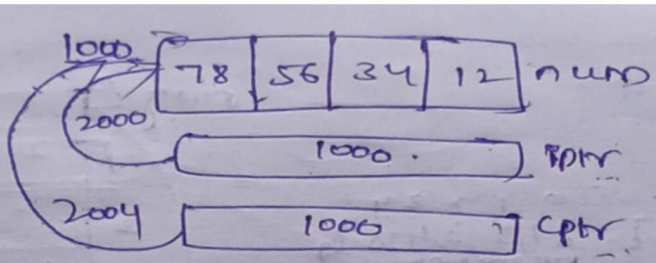


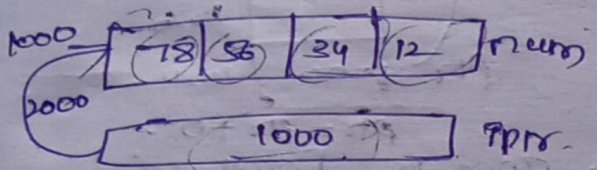
for little Endian: LSB in lower memory address



Example:

```
#include<stdio.h>
int main()
{
    int num = 0x12345678;
    int *iptr, char *cptr;
    iptr = &num;
    cptr = &num;
}, return 0;
```

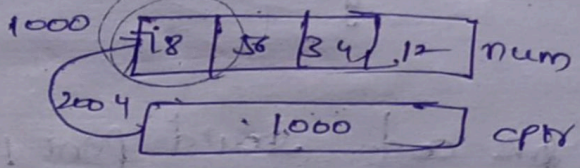




*fptr = 0x12345678.

↓

It takes 4 bytes



*cptr = 0x78.

↓

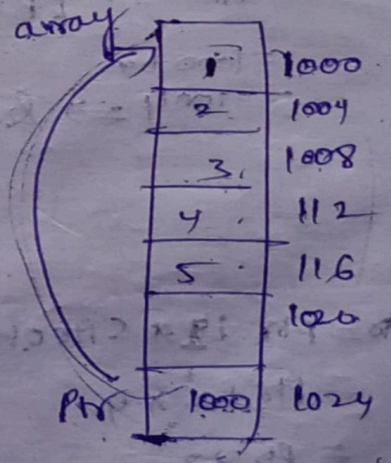
It take only one byte

Rule 5: Pointer Arithmetic:

Value(p+i) = Value(p) + i * sizeof(*p)

Example:

```
#include<stdio.h>
int main()
{
    int array[5] = {1,2,3,4,5};
    int *ptr = array;
    return 0;
}
```



So, Address of array = 1000.

Base address = 1000

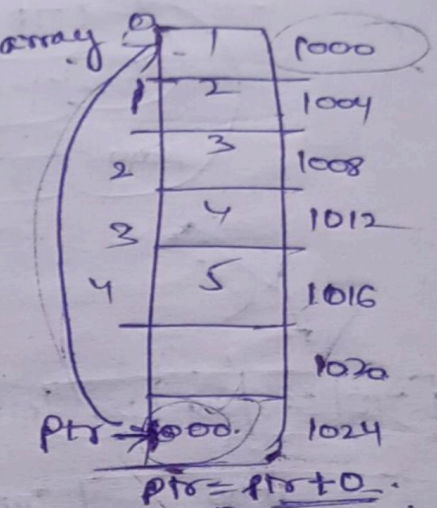
&array[0] = 1 → 1000

&array[1] = 2 → 1004

Examples:

```
#include<stdio.h>
int main()
{
    int array[5] = {1,2,3,4,5};
    int *ptr = array;
    printf("%d\n", *ptr);
    return 0;
}
```

(array indices: 0 1 2 3 4; addresses: 1000 1004 1008 1012 1016)



→ This code should print 1 as output since its points to the base address.

Now, what we do is, ptr = ptr + 1;

The above line can be described as follows:

ptr = ptr + 1 * size of (data type).

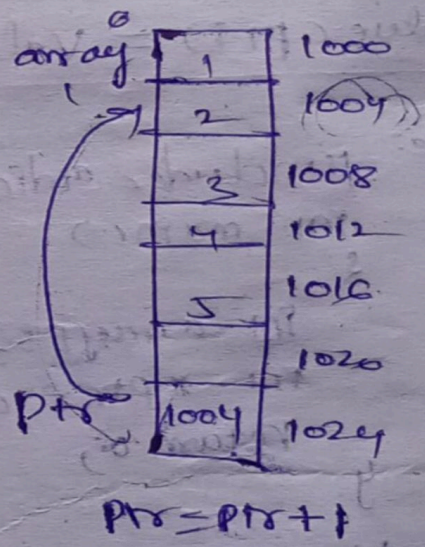
Ex: ptr = ptr + 1 * size of (int).

= ptr + 1 * 4

= ptr + 4. [ptr = 1000]

= 1000 + 4

= 1004 ⇒ &array[1]



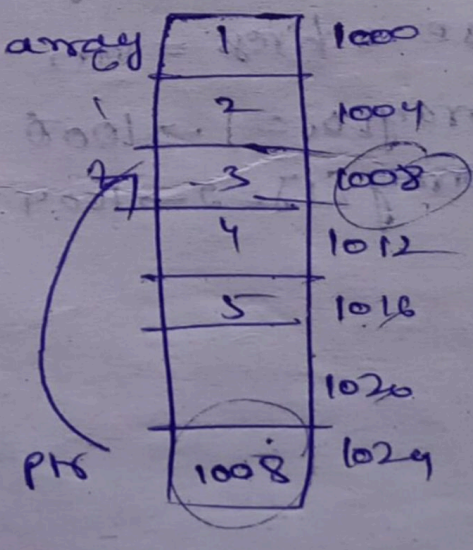
Ex: ptr = ptr + 2 * size of (int)

= ptr + 2 * 4

= ptr + 8

= 1000 + 8

= 1008 ⇒ &array[2]

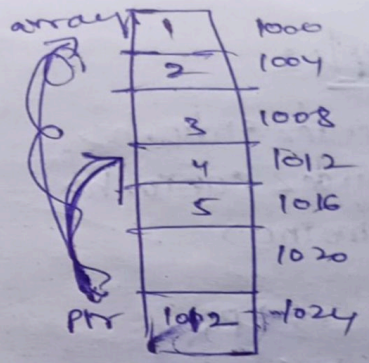


ex: Ptr = ptr + 3 * size of (int)

= ptr + 3 * 4

= 1000 + 12

= 1012 ⇒ &array[3]



Ptr = Ptr + 3

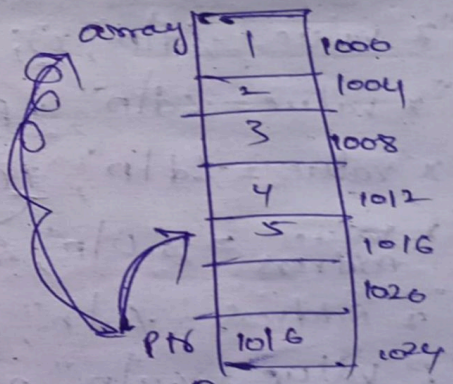
ex: Ptr = ptr + 4 * size of (int)

= Ptr + 4 * 4

= Ptr + 16

= 1000 + 16

= 1016 ⇒ &array[4]



Ptr = ptr + 4

Null Pointer:

* If we declare one uninitialized pointer, it will be pointing some random address.
* Again if we initialize the value in that pointer leads to an undefined behaviour.
* If we want to dereference the uninitialized pointer, it leads to segmentation fault.

Ex: int *ptr; // garbage address will be there.
*ptr = 10; // undefined behaviour

So, its better to initialize the pointer with Null.

⇒ dereferencing the null pointer also gets segmentation fault.

Ex:
```
#include<stdio.h>
int main()
{
    int *ptr=10;
    printf("%d", *ptr);
}
```

Ptr = 10
*ptr ⇒ *10

It will pointing to the invalid memory address. Illegal memory access will happen.
So, the o/p is segmentation fault.

```
int *ptr;
*ptr=10;
printf("%d", *ptr);
```
} segmentation fault

Ex:
```
main()
{
    int *ptr=10;
    printf("%d", ptr);
}
```
o/p: 10.

Ex:
```
main()
{
    int *ptr;
    ptr=10;
    printf("%d", ptr);
}
```
o/p: 10.

⇒ The representation of Null Pointer is "(void *) 0" or other address depending on operating system.

⇒ The pointers which have Null stored in them are called as Null pointers. These pointers will give the fixed result when dereferenced. i.e., Segmentation fault.

Void Pointers

Void Pointer is nothing but the generic Pointer.

~~Void Pointer~~ Syntax: `void *vptr;`

⇒ Void Pointers cannot be deferenced. If we deference it. It leads to compiler error.
Because, the compiler does not what kind of data should be fetched.
So, typecasting is necessary before dereferencing the Void Pointer.

⇒ *(int *)vptr ⇒ This means that whatever the address vptr ~~we are storing~~ holding should be treated as an integer.

⇒ Pointer arithmetic on void Pointer is "compiler implementation dependent".

⇒ In gcc, sizeof(void) is "1."

(sizeof void *) ⇒ 4/8 Bytes.

Examples:

```
#include<stdio.h>
int main()
{
    int num = 0x12345678;
    void *ptr = &num;
    pf("%d ", *(int *)(ptr+1)); // Giv
    pf("%d ", *(int *)(ptr+1)); // Giv
}
```

num = | 12 | 34 | 56 | 78 |
100 104 108 112

*(int *)(100+1*sizeof(void))
*(int *)(100+1+1)
*(int *)(101) ⇒ bus error (Because 101 is not an valid address for integer).

*(int *)(ptr+1)
*(100+1*sizeof(int))
*(100+1*4) ⇒ *(1004)

⇒ If we go with Integer Pointer, our intension is store the address of Integer variable.

⇒ If we go with void Pointer, address of any given variable should be stored.

* Before the typecasting, the pointer arithmetic is based on sizeof(void).
* After typecasting, the pointer arithmetic is based on sizeof(datatype)

* Generic swap program using integers:

```
#include<stdio.h>
void swap(void *ptr1, void *ptr2, int size);
int main()
{
    int num1, num2;
    Pf("Enter the num1: ");
    Sf("%d", &num1);
    Pf("Enter the num2: ");
    Sf("%d", &num2);
    swap(&num1, &num2, sizeof(int));
    Pf("%d %d", num1, num2);
    return 0;
}
void swap(void *ptr1, void *ptr2, int size)
{
    int i;
    char temp;
    for(i=0; i<size; i++)
    {
        temp = *(char *)(ptr1+i);
        *(char *)(ptr1+i) = *(char *)(ptr2+i);
        *(char *)(ptr2+i) = temp;
    }
}
```

O/p: Enter the num1: 10
Enter the num2: 20
20 10

num1: 00 00 00 10

Little Endian:

10 00 00 00

~~* Generic swap prog~~

10

0	1	2	3
0000 1010	0000 0000	0000 0000	0000 0000

20

0001 0100	0000 0000	0000 0000	0000 0000

After swap:

0001 0100	0000 0000	0000 0000	0000 0000
0000 1010	0000 0000	0000 0000	0000 0000

Generic swap program using characters:

```
#include <stdio.h>
void swap(void *ptr1, void *ptr2, char size);
int main()
{
    char num1, num2;
    printf("Enter the num1: ");
    scanf("%c", &num1);
    printf("Enter the num2: ");
    getchar();
    printf("Enter the num2: ");
    scanf("%c", &num2);
    swap(&num1, &num2, sizeof(char));
    printf("%c %c", num1, num2);
    return 0;
}
void swap(void *ptr1, void *ptr2, char size)
{
    int i, temp;
    for(i=0; i<size; i++)
    {
        temp = *(char *)(ptr1+i);
        *(char *)(ptr1+i) = *(char *)(ptr2+i);
        *(char *)(ptr2+i) = temp;
    }
}
```

Enter the num1: a
Enter the num2: b
b a

Dynamic Memory Allocation:

Static memory:

1) In static memory, once we declare the size in an array. It can be fixed.
2) It is not possible to extend the size in static memory.
3) Here, compiler will decide the memory.
4) It is an "named memory location".
5) Here, the memory will be allocated in stack segment.

Disadvantages:

1) shortage of memory.
2) wastage of memory.

Dynamic memory: Here, the memory will depend on the user.

* for this, the memory will be allocated in heap segment.
* Dynamic memory can be modified, extended (or) deleted whenever we require.
* Dynamic memory is managed with the help of pointers like malloc, calloc, realloc and free.
* these are an "unnamed memory" location, to control them we need pointers.
* All these functions are part of "stdlib.h".

Example:

```
int main()
{
    // dynamic memory allocation 100 byte.
    // some operation.
    // free the memory.
}
```

1) **Malloc:**

→ malloc means continous memory allocation.

syntax:

```
void * malloc(size_t size);
```

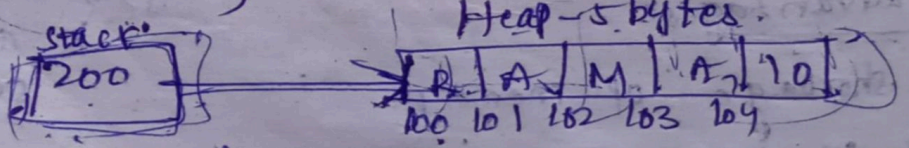
Return types "void".

unsigned int (or) unsigned long int.

→ If the continous memory is not available, Then malloc will return NULL.

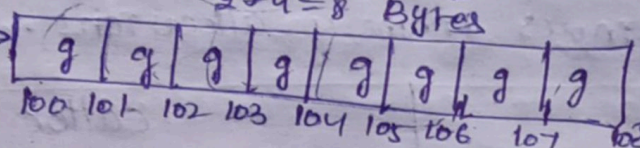
Example:

1)
```
main()
{
    int *ptr;
    ptr = malloc(5); // Implicitly typecasting from void * to int *.
    strcpy(ptr, "RAMA");
}
```



2) ptr = malloc(2 * sizeof(int));

2*4 = 8 Bytes



Example: #include<stdio.h>
#include<stdlib.h>
int main()
{
int *ptr = malloc(sizeof(int));
*ptr = 10;
Pf("Value is: %d", *ptr);
return 0;
}

O/P: Value is: 10.

main()
{ char *ptr;
ptr = malloc(5);
return 0;
}

2) calloc:

→ calloc is also a continous memory allocation.

Syntax:

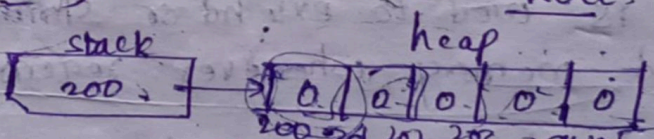
void * calloc(size_t nmemb, size_t size);

where nmemb ⇒ Number of blocks (or) Number of elements.
size ⇒ size of each member.

Example:

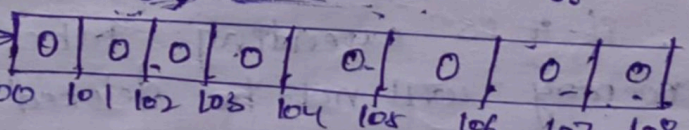
main()
{ char *ptr;
ptr = calloc(5,1);
}

⇒ calloc will always seach for 5 bytes of continous memory in the heap. If it is not available it will returns "NULL"



Example:

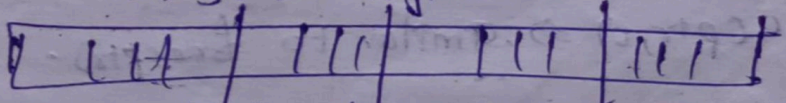
Ptr = calloc(2, sizeof(int));



(or)

calloc(3, sizeof(int))

⇒ 3*4 = 12 Bytes



read the data block by block.

Differences b/w malloc & calloc:

1) Both are continous memory allocations.

malloc	calloc.
2) malloc have only one Parameter.	2) calloc have two parameters.
3) The default value of malloc is Garbage value.	3) The default value of calloc is '0'.
4) It will allocate the memory byte by byte.	4) It will allocate the memory block by block.
5) Execution speed is faster	5) Execution speed is slower.
6) Malloc is better with structure and basic memory allocation.	7) calloc is better with array because of initialization to 0.

Both will get the memory in heap segment.

3) realloc():

⇒ It is used for extending (or shrinking) the memory.

⇒ Realloc is used to extend or shrink the previously allocated memory whenever required. → using malloc (or) calloc.

Syntax: `void * realloc(void *ptr, size_t size);`

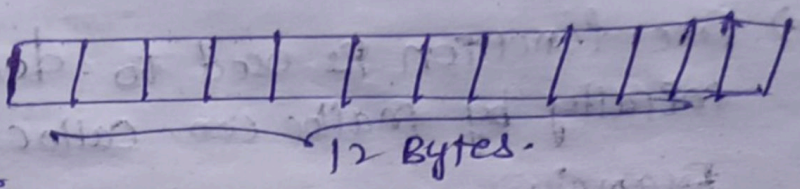
⇒ Realloc is similar to malloc, if it is shrinking no initialization. If it is extending takes garbage values. But, the memory will not be deleted.

Example:

```
ptr=NULL;
ptr=realloc(ptr,10) → similar to malloc(10);
realloc(ptr,0) ⇒ similar to free(ptr).
```

Examples: #include<stdio.h>

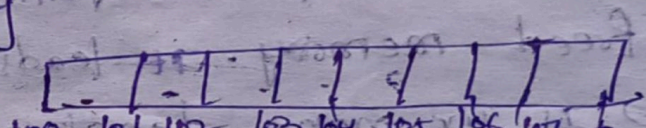
```
int main()
{
    int *ptr;
    ptr = malloc(12);
    ptr = realloc(ptr, 8); // Here, we shrinking the memory. the remaining 4 bytes takes Garbage values. It will not be deleted.
    ptr = realloc(ptr, 20); // Here, we extended the 12 bytes of memory. these 12 bytes takes Garbage values.
}
```



Example:

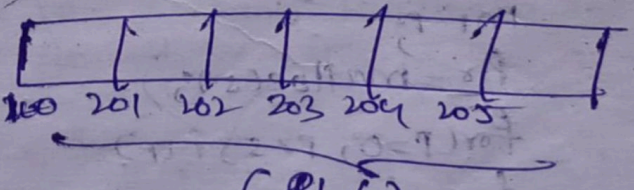
```
#include<stdio.h>
int main()
{
    char *p1 = malloc(5);
    char *p2 = malloc(3);
    realloc(p1, 6);
}
```

for this the memory will allocated continously.



Here, we want to add one extra byte to p1. But, p1 & p2 are continously allocated. so, it's not possible to allocate that one byte after p1. so, that ptr bytes memory, where the memory will be shifted to the new continously pt will allocate from p1.

This, the previous 5 bytes are deleted.



⇒ If realloc() fails to extend/expand memory as requested. Then it returns 'NULL', the data on the old memory remains unaffected.

4) free():

⇒ free function is used to deallocate the memory dynamically by malloc (or) calloc function.

Syntax: `void free(void *Ptr);`

⇒ After allocating the memory, if is free. If we want to allocate some more to this without free. It leads to memory leakage.

↓

Improper usage of the memory leads to memory leak.

⇒ If free is not called after dynamic memory, will leads to memory leak.

⇒ Although heap is deleted from the function, the address which is freed will be available in the pointer.

⇒ Still the pointer pointing (or) holds the already freed memory. It leads to dangling pointer.

⇒ If we make dangling pointer to point to NULL, undefined behaviour will be avoided.

⇒ In free function, compulsory we can pass the address. other wise it leads to memory management mechanism.

free(); → MMech

Example:

```
#include<stdio.h>
int main()
{
    char *Ptr;
    int i;
    Ptr = malloc(5);
    for(i=0; i<5; i++)
    {
        Ptr[i] = 'A' + i;
    }
    free(Ptr);
    return 0;
}
```

Ptr → 1000

malloc(5):

'A'	'B'	'C'	'D'	'E'
101	102	103	104	105

Ptr[0] = 'A' + 0 ⇒ 65 ⇒ A

Ptr[1] ⇒ 'A' + 1 ⇒ 66 ⇒ B

Ptr[2] ⇒ 'A' + 2 ⇒ 67 ⇒ C

Ptr[3] ⇒ 'A' + 3 ⇒ 68 ⇒ D

Ptr[4] ⇒ 'A' + 4 ⇒ 69 ⇒ E

CONST keyword.

Qualifiers.

* The qualifiers are keywords which are applied to the data types.

Qualifiers are of two types:

(i) const.
(ii) Volatile.

(i) Const: It is a keyword applied on a variable.
This is to tell the compiler, that the value will not be changed throughout the program.

→ const int num & int const num: these both are same.
num is an integer constant.

Example:

```
int main()
{
    const int num = 20;
    num = 30; //error;
    return 0;
}
```

→ Here, address is constant number

→ const int *ptr & int const *ptr: ptr is the pointer to an constant integer. changing *ptr is not allowed.

→ Here, value is constant

Example:

```
int main()
{
    const int num = 10;
    const int *ptr = &num;
    *ptr = 10; //error
    (*ptr)++; //error
    return 0;
}
```

ptr →

→ int *const ptr: ptr is a constant pointer to an integer. changing ptr is not allowed. →these, address is constant.

Example:

```
int main()
{
    int num1, num2 = 20;
    int *const ptr = &num1;
    ptr = &num2; //error.
    ptr++; //error
    return 0;
}
```

num1 = 10

→ const int * const ptr: ptr is a constant pointer to an integer. Changing ptr ~~is not allowed~~ and *ptr is not allowed.

Example:
```
int main()
{
    int num1=10, num2=20;
    const int * const ptr = &num;
    ptr++; //error.
    (*ptr)++; //error.
    ptr = &num2; //error.
    *ptr=20; //error.
    return 0;
}
```

Bitwise operators are not allowed in pointers. It allows only logical operators.

Pitfalls of pointers.

1) Segmentation fault:

→ It is a type of Runtime error.
→ This is caused whenever we are accessing the address that is not allowed to be accessed.

Example:
```
main()
{
    int *ptr=10;
    Pfc"%d", *ptr);
}
```

ptr = [10] (10000)

*ptr ⇒ *10 ↓ This is not an valid address. so, for accessing address 10 is not allocated. there, memory leak will happen.

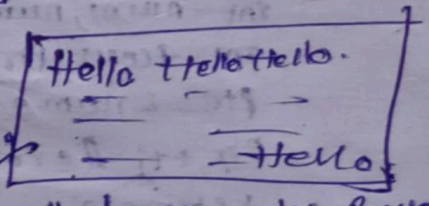
O/P: segmentation fault.

→ when stack overflow may occur, it leads to segmentation.
⇒ If we dereference the uninitialized the pointer, segmentation fault may occur.

Example:
```
main()
{
    while(1)
    {
        Pfc"Hello");
        sleep(1);
    }
    return 0;
}
```

<unistd.h>

Infinite times execute



Buffer filled - at only Buffer Overflow may occur. It leads to segfault

2) Dangling Pointers -

→ The pointer is pointing to a already freed location is called as Dangling Pointer.

→ Dereferencing the dangling pointer leads to undefined behaviour.

Example:
```
int main()
{
    int *ptr = calloc(5,1);
    free(ptr);
    printf("%d\n", *ptr); // ptr is a dangling pointer.
    return 0;
}
```
o/p: 0.

Here, ptr is still having the address which it was pointing to. But the address is no longer with the user. as it has already been freed.

3) Undefined behaviour: The behaviour of the variable cannot be define anything. i.e., which cannot be define anything.

→ If we return the address of the local variable leads to undefined variable.

→ If we do the post increment with same variable multiple times in printf() leads to undefined behaviour.

→ If we are trying to free the already freed memory. leads to undefined behaviour.

→ Dereferencing the void pointer leads to undefined Behaviour.

→ Dereferencing the dangling pointer leads to undefined behaviour.

4) Wild Pointers

→ An uninitialized pointers which are pointing to some random address is called as ~~dangling~~ wild Pointer.

→ So, good practice is to initialize it with NULL when the pointer is being declared.

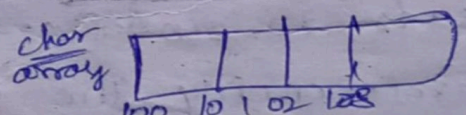
Example: main()
{
int *ptr; // void pointer
static int *ptr1; // not a void pointer.
Because, static pointers are initialized with Null by default.
return 0;
}

5) Memory leaks:
→ Failing to free the dynamically allocated memory leads to memory leak.
→ Memory leakage, means improper usage of the Memory.

Example: int main()
{
int *ptr;
while(1)
{
ptr = malloc(100);
// forgot to free the memory;
}
return 0;
}

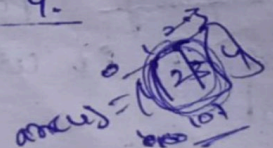
6) Bus error: This error is caused when the cpu cannot handle the address in the address bus.

Example: main()
{
char array[4];
int *ptr = &array[1];
scanf("%d\n", ptr); // bus error
return 0;
}



ptr = 101
Here, trying to read an integer to the address which is not a multiple of 4. so bus error will come.

⇒ usually the cpu can handle the address which is a multiple of 4.



2D Array

→ 2D Array is a collection of rows & columns.

Syntax: `datatype array_name[rows][columns];`

→ outer loop represents the rows and inner loop represents the columns. ie:) Nested loop.

→ Specifying the columns is compulsory.

inner loop → to trace the columns of each row.

`Total bytes = rows * columns * sizeof(datatype);`

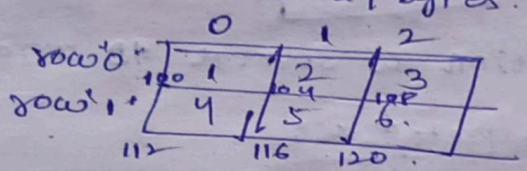
Example:

```
int arr[2][3] = { 1,2,3,4,5,6};
```

(or)

```
int arr[2][3] = {{1,2,3},{4,5,6}};
```

Total bytes = 2 * 3 * 4
= 24 bytes.



arr[0][0] → 1
arr[0][1] → 2
arr[0][2] → 3
arr[1][0] → 4
arr[1][1] → 5
arr[1][2] → 6.

Example:

```
#include<stdio.h>
int main()
{
    int size1, size2, i, j;
    printf("Enter the rows: ");
    scanf("%d", &size1);
    printf("Enter the columns: ");
    scanf("%d", &size2);
    int arr[size1][size2];
    for(i=0; i<size1; i++)
    {
        for(j=0; j<size2; j++)
        {
            scanf("%d", &arr[i][j]);
            printf("%d", arr[i][j]);
        }
    }
    return 0;
}
```

O/p: Enter the rows: 2
Enter the columns: 2
1 2 3 4 1 2 3 4.

For 1D array,

Printing the elements ⇒ *(a+i).
reading the elements ⇒ (a+i).

For 2D array,

Printing the elements ⇒ *(*(a+i)+j)

Explanation:

a[i][j] ⇒ x[j] ⇒ *(x+j)

x[i] = x. ⇒ *(a[i]+j) a[i] → *(a+i)

⇒ *(*(a+i)+j).

So a[i][j] → *(*(a+i)+j)

⇒ 2D array is a combination of several 1D array.

The 2D array can be interpreted as:

(i) arr[i][j]
(ii) *(arr[i]+j)
(iii) *(*(arr+i)+j).

Compiler will allocate the memory internally at the time of definition of variables

Advantage of Dynamic memory:

(i) we can decide size of memory at run time.
(ii) we can resize it whenever require.
(iii) we can decide when to create and destroy.

Static:

```
main()
{
    char a[100];
    return 0;
}
```

Dynamic:

```
main()
{
    char *ptr;
    Ptr = malloc(5);
    return 0;
}
```

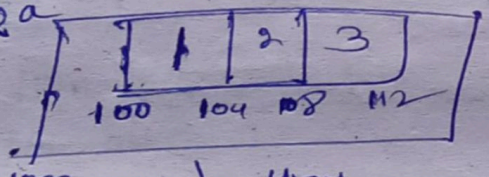
Multilevel Pointer: It stores the address of another pointer. Using this we can perform array of pointers, pointer to an array etc.

Example: main()
{
int num=10;
int *ptr1=#
int **ptr2=&ptr1;
int ***ptr3=&ptr2;
Pfc"%d", ptr3); //3000
Pfc"%d", *ptr3); //2000
Pfc"%d", **ptr3); //1000
Pfc"%d", ***ptr3); // 10.
}

num [10] 1000
ptr1 [1000] 2000
ptr2 [2000] 3000
ptr3 [3000] 4000

ptr3 → 3000
*ptr3 → *3000 → 2000
**ptr3 → **3000 → *2000 → 1000
***ptr3 → ***3000 → **2000 → *1000 → 10.

For 1D array:



1000 → whole address.

1000 → 1st rows address ⇒ a
100 ⇒ 1st row 1st element address ⇒ *a.
1 ⇒ 1st element ⇒ **a.
&a → whole array.

Example:
main()
{
int a[3]={1,2,3};
Pfc"%p\n", a); //1000
Pfc"%p\n", &a[0]); //1000
Pfc"%p\n", &a); //1000
}

Example:
main()
{
int a[3]={1,2,3};
Pfc"%p\n", a+1);
Pfc"%p\n", &a[0]+1);
Pfc"%p\n", &a+1);
100+1*4 ⇒ 1004
100+1*4=1004
100+1*12 ⇒ 112

* Array of pointers: It is an array which holds the addresses.
→ Array of pointers is a collection of address.

Generalized formula: ++(ptr+i)

Syntax: datatype *pointer_name[size];

Eg: *ptr[i]

Ex: int *ptr[3]; // This pointer stores the address of 3 variables.

Total memory will be dependent on bitness of the system:

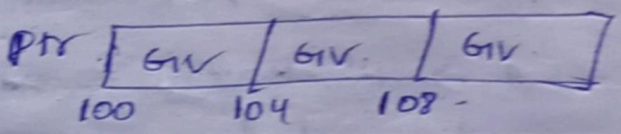
(i) 32 bit system:

total memory = size * sizeof(pointer)
= 3 * 4
= 12 bytes

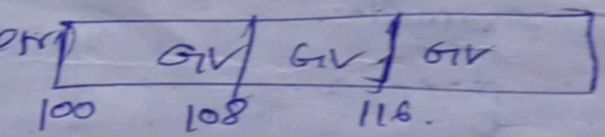
(ii) 64 bit system:

total memory = 3 * 8
= 24 bytes.

memory layout for 32 bit:

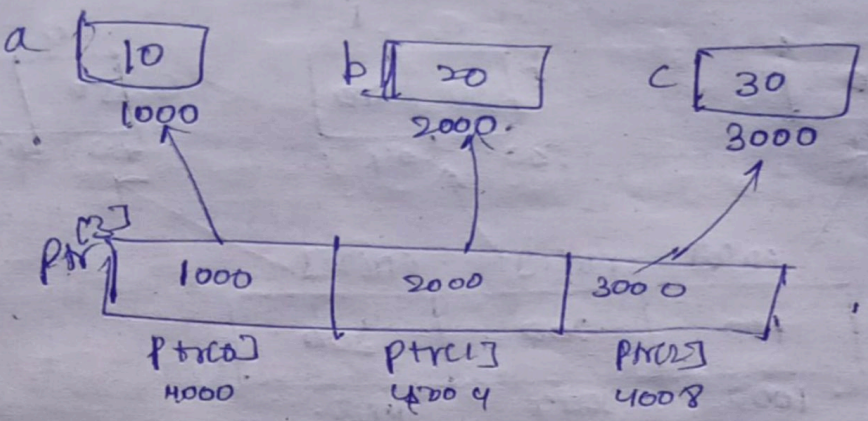


Memory layout for 64 bit:



Initializing and accessing of array of pointer elements:

1) main()
{
int a=10, b=20, c=30;
int *ptr[3];
ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;
(OR) int *ptr = {&a, &b, &c};
0 1 2
printf("%d", ++(ptr+0));
return 0;
}



$\Rightarrow$ *(ptr+0) $\Rightarrow$ *(4000+0*4)

*ptr[0] $\Rightarrow$ *(4000) $\Rightarrow$ 1000 $\Rightarrow$ 10.

$\Rightarrow$ *(ptr+1) $\Rightarrow$ *(4000+1*4)

*ptr[1] $\Rightarrow$ *(4004) $\Rightarrow$ 2000 $\Rightarrow$ 20.

→ Array of pointers can also holds the address of two (or) more arrays.

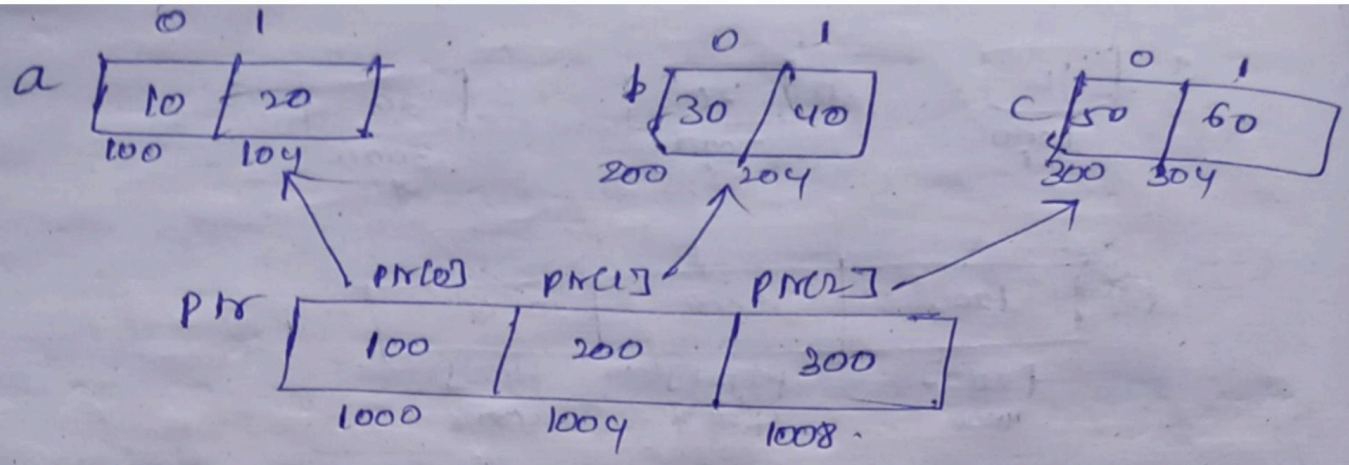
Example:

```
#include<stdio.h>
int main()
{
    int a[2] = {10,20};
    int b[2] = {30,40};
    int c[2] = {50,60};
    int *ptr[3];
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<2; j++)
        {
            ptr[0] = &a;
            ptr[1] = &b;
            ptr[2] = &c;
            ptf("%d ", *(*(p+i)+j));
        }
    }
}
```

(or)

```
main()
{
    int a[2] = {10,20};
    int b[2] = {30,40};
    int c[2] = {50,60};
    int *ptr[3] = {a,b,c};
}
```

Op: 10 20 30 40 50 60.



⇒ *(*(ptr+0)+0)) ⇒ *(*(1000)) ⇒ *(*100) ⇒ 10

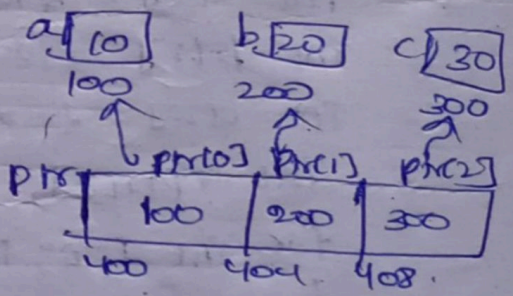
⇒ *(*(ptr+2)+1)) ⇒ *(*(1000+2*4)+1*4)) ⇒ *(*(1008)+4) ⇒ *(300+4) ⇒ *(304) ⇒ 60.

Passing array of pointer to function:

⇒ Here, we can collect the variable on multiple level pointer.

Example:

```
#include<stdio.h>
void print_array(int **p)
{
    int i;
    for(i=0; i<3; i++)
    {
        printf("%d ", *p[i]);
        or
        **(p+i)
    }
}
int main()
{
    int a=10, b=20, c=30;
    int *ptr[3] = {&a, &b, &c};
    print_array(ptr);
    return 0;
}
```



→ *p[0] ⇒ *100 ⇒ 10.

→ *p[1] ⇒ *200 ⇒ 20.

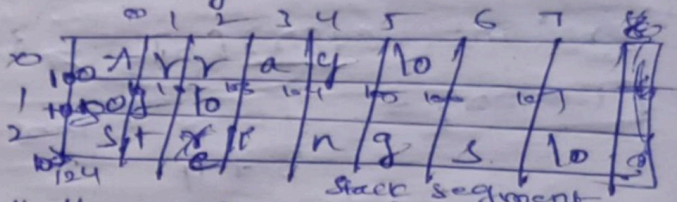
→ *p[2] ⇒ *300 ⇒ 30.

Array of strings: In a single array, we can store multiple strings.

→ Array of string is a collection of strings, which is a 2 dimensional array.

→ First dimension represents ⇒ how many strings there are.
second dimension represents ⇒ the maximum length of each string.

Ex: char s[3][8] = { "Array", "of", "Strings" };
(3 ↙ number of strings; 8 ↘ length of each string.)



Ex: main()
{
char [3][8] = { "Array", "of", "Strings" };
Pf("%s %s %s\n", s[0], s[1], s[2]);
return 0;
}

~~s[2][1] = 'e';~~

s[1] ⇒ *(s+1 * sizeof(s[0]))
⇒ *(100 + 1 * 8)
⇒ *108 ⇒ of.

⇒ *s[2][1] = 'e' ⇒ *(*(s+2 * sizeof(char[8])) + 2 * sizeof(char))
⇒ *(*(100 + 2 * 8) + 2 * 1)
⇒ *(*(100 + 16) + 2)
⇒ *(*(124) + 2)
⇒ *126 ⇒ e

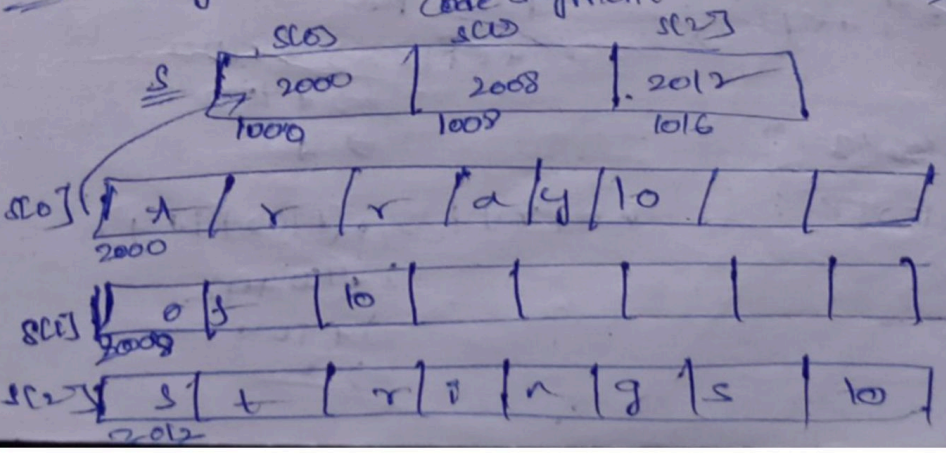
Ex: ~~Array of strings using~~ Array of Pointers:

main()
{
char *s[3]; //3 * 8 = 24 Bytes
s[0] = "Array";
s[1] = "of";
s[2] = "Strings";
Pf("%s", s[0]);
}

Olp: Array.

→ Memory is present in code segment.

malloc



s[0]
⇒ *(s+0)
⇒ *(1000 + 0 * 1)
⇒ *(1000)
⇒ 2000
⇒ Array.

Ex: main()
{
char s[3][8] = {"Array", "of", "strings"};
for(int i=0; i<3; i++)
{
for(int j=0; j<8; j++)
{
printf("%c", s[i][j]);
}
}
return 0;
}

O/p: Array of strings.

* **Pointer to an array:** It is a pointer which holds the whole address of an array.

Syntax: `datatype (*ptr_name)[size];`

Ex: `int (*ptr)[3];` → here, if we miss the parantheses it leads to compiler error.

memory = size * sizeof(datatype)
= 3 * 4
= 12 Bytes

→ Here, ptr is a pointer which is pointing to array of 3 integers.

→ If we dereference the pointer, whole array size will be calculated.

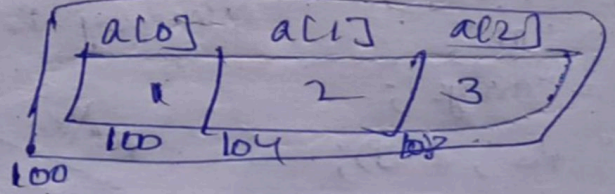
→ Pointer arithmetic on pointer to an array will be;

ptr+1 = ptr + 1 * sizeof(1D array)

Generalized formula: *(*ptr+i)

Example:

```
int main()
{
    int arr[3] = {1, 2, 3};
    int (*ptr)[3];
    ptr = &arr;
    printf("%d\n", **ptr);
    return 0;
}
```



ptr = 100

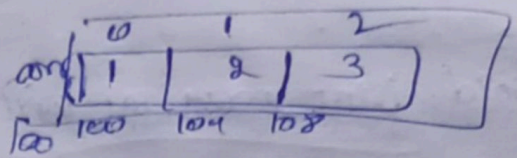
**ptr ⇒ *100
⇒ *100
⇒ 1

**ptr+1 ⇒ *(*100+1*4)
⇒ *(100+4)
⇒ *104
⇒ 2

**ptr+2 ⇒ *(*100+1*4)
⇒ *(100+8)
⇒ *108
⇒ 3

Ex: #include<stdio.h>
int main()
{
int array[3] = {1,2,3};
int (*ptr)[3];
for(int i=0; i<3; i++)
{
ptr = &arr;
Pf("%d\n", *(*ptr+i));
}
return 0;
}

o/p: 1
2
3.



ptr = 100.

⇒ *(*ptr+0) ⇒ *(*100+0)
⇒ *(100+0)
⇒ 1

⇒ *(*ptr+1) ⇒ *(*100+1*4)
⇒ *(100+4)
⇒ *104
⇒ 2

⇒ *(*ptr+2) ⇒ *(*100+2*4)
⇒ *(100+8)
⇒ *108
⇒ 3.

⇒ we cannot create the array in the function parameters like fun(int a[]).

Passing 2D array to function:

1) the way array is declared as,
we can pass 2D array as a way it is declared like,

```
void print_array(int arr[2][3]);
```

2) Pointer to an array:
⇒ one way of passing 2D array to a function is pointer to an array.

```
void print_array(int (*ptr)[3]);
```

3) Array of pointers:
2D arrays are also passed by using array of pointers.

```
void print_array(int *ptr[]);
```

4) By passing size along with array address;
⇒ passing 2D array along with no. of rows & columns.

```
void print_array(int row, int col, int arr[row][col]);
```

⇒ If the order of the arguments are change, it will give error.

ex:
void print_array(int arr[row][col], int row, int col);
compile time error.

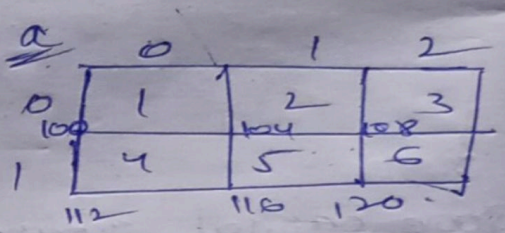
5) By normal integer pointer,

void print_array(int row, int col, int *ptr);

Examples:

```
#include<stdio.h>
void print_array(int p[2][3]);
{
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}
int main()
{
    int a[2][3] = {{1,2,3},{4,5,6}};
    print_array(a);
    return 0;
}
```

a → row address.



o/p: 1 2 3 4 5 6

* Pointers – 2D Array creations:

Each dimension should be static or dynamic.

1) Both static (C Rectangular array): (CBS):

It means, both rows and colum are static.

Example: int a[2][3];

int a[rows][cols];

static → array.
dynamic → pointer

→ this is an 2D array

2) First static second dynamic (FSSD):

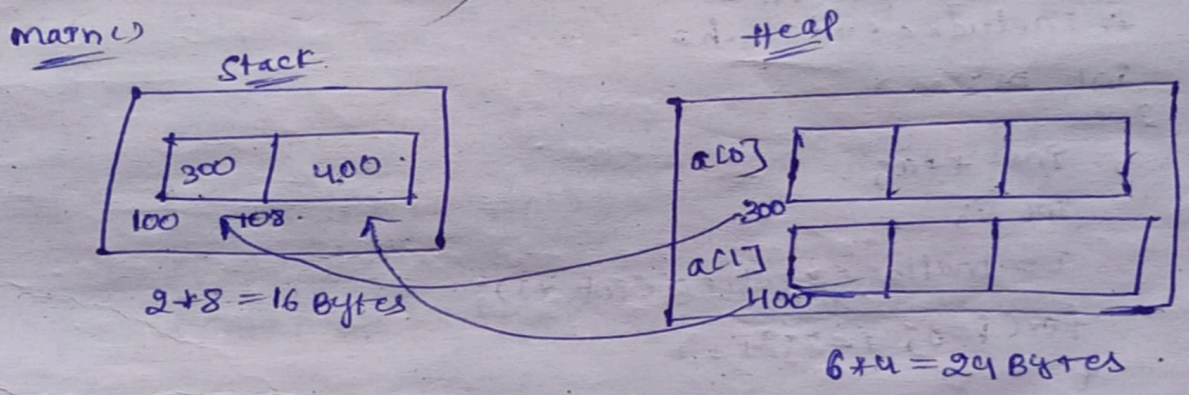
this means, row static and column dynamic.

→ It represents "Array of pointers".

Example: `int *a[rows];`

Example:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *a[2];
    for(int i=0; i<2; i++)
    {
        a[i] = malloc(3*sizeof(int));
    }
    return 0;
}
```



⇒ 16 + 24 ⇒ 40 Bytes.

3) First dynamic second static (FDSS):

This means, row dynamic and column static.

→ It represents "pointer to an array".

Example: `int (*p)[3];` ← size of row.

Example:

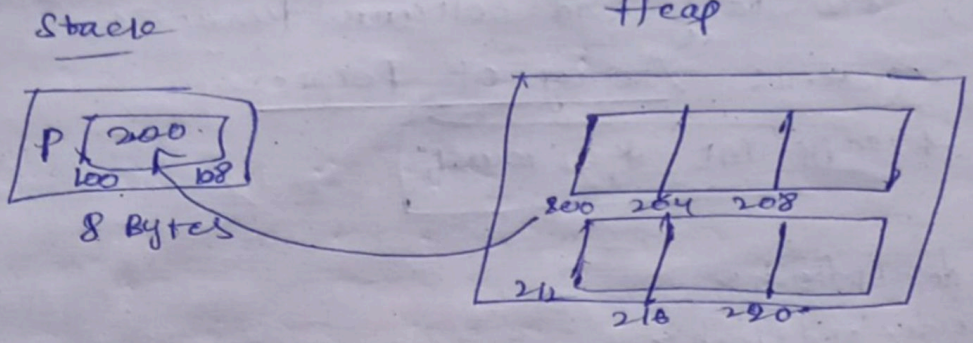
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int (*p)[3];
    p = malloc(2*sizeof(int[3]));
    return 0;
}
```

3*4

2*12 ⇒ 24 Bytes.

→ Malloc have starting address. → malloc have continous memory allocation.

Stack



total: 8+24 ⇒ 32 Bytes.

4) Both dynamic:

This means, Both row & column are dynamic.

→ It represents "multi level pointer".

Example: `int **a;`

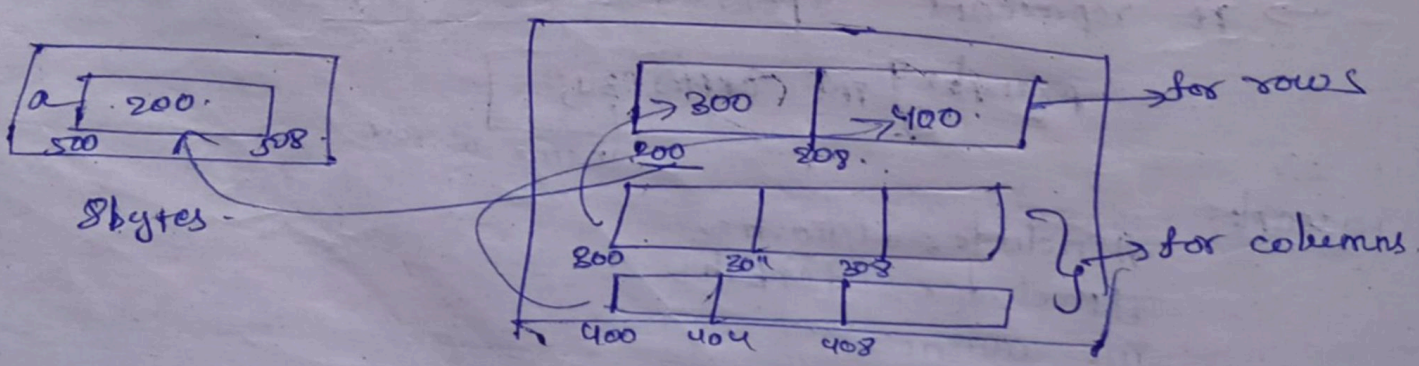
Example:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int **a;
    int i;
    a = malloc(2*sizeof(int *));
    for(i=0; i<2; i++)
    {
        a[i] = malloc(3*sizeof(int));
    }
    return 0;
}
```

main()

Stack

Heap



2+8 + 3*4 + 3*4 ⇒ 16+12+12.

2 rows

8+16+12+12 ⇒ 48 bytes.

function pointers.

⇒ The pointer which holds the address of the function is called as "function pointer".

Syntax:

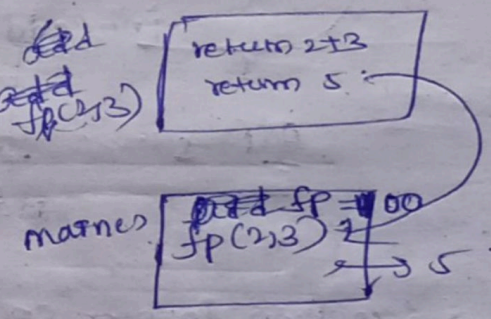
```
return_type (*fp) (list of arguments(s) datatype);
```

Here, we can pass any type of arguments.

*fp ⇒ It is a pointer which can holds the address of the function.

Example:

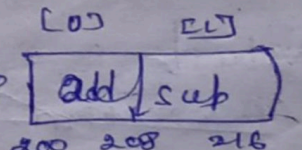
```
#include<stdio.h>
int add(int num1, int num2)
{
    return num1+num2;
}
int main()
{
    int (*fp)(int,int);
    fp=add;
    printf("%d", fp(2,3));
}
```



O/P: 5.

Example: Array of function pointers: It is an array which holds the address of the function.

```
#include<stdio.h>
int add(int num1, int num2)
{
    return num1+num2;
}
int sub(int num1, int num2)
{
    return num1-num2;
}
int main()
{
    int (*fp[])(int,int)={add,sub};
    printf("%d\n", fp[0](2,4));
    printf("%d\n", fp[1](2,4));
}
```



fp[0] fp[1]

O/P: 6
-2

Call Back functions

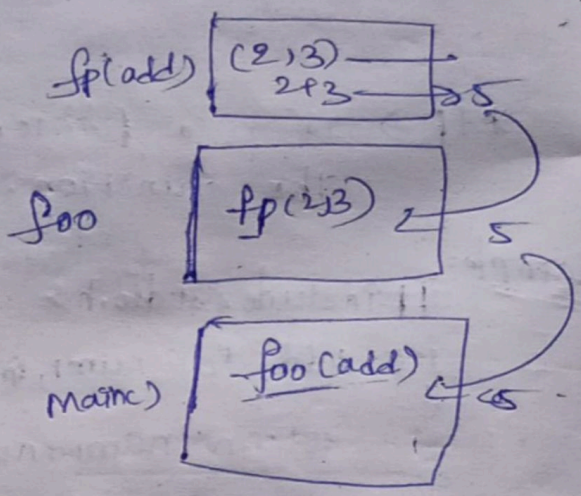
Passing the address of another function to a function is called "call back function".

Example:

```
#include<stdio.h>
int foo(int(*fp)(int, int))
{
    int res = fp(2,3);
    printf("%d", res);
}
int add(int a, int b)
{
    return a+b;
}
int main()
{
    foo(add);
}
```

O/P: 5

address of another function.



⇒ function pointer contains the address of the code.
⇒ Normal pointer contains the address of the data.
⇒ In function pointer, one function can call all the functions.

Volatile keyword

→ Volatile is a keyword. It is used to remove the code optimization.
→ It instructs the compiler, the code is not optimized.
⇒ volatile keyword is a qualifier, that is applied to a variable when it is declared.
⇒ volatile keyword "always read from memory".
⇒ It tells the compiler that the value of the variable may change at any time, without any action being taken by the code.

Examples:

```
#include<stdio.h>
int main()
{
    volatile int i; // do not optimize the code, whenever
                    i is being used.
    while(1)
    {
        led_on();
        //delay
        for(i=0; i<1000; i++);
        led_off();
        //delay
        for(int i=0; i<1000; i++);
        i=100;
    }
}
```

If these delays are not there, we can't get the correct o/p. so, we can't optimize this.

Ex:

```
#include<stdio.h>
int main()
{
    volatile int i;
    for(int i=0; i<10; i++);
    {
        printf("%d ", i);
    }
}
```

O/p: 10.

In this, for loop having semi column. It iterates continuously till the condition getting false.

As our compilers are smart compilers, they should avoid the unnecessary code blw them. But, volatile keyword tells the compiler, code should not be optimized.

Ex:

```
#include<stdio.h>
int main()
{
    volatile int i;
    for(int i=0; i<10; i++)
    {
        printf("%d ", i);
    }
}
```

O/p: 0 1 2 3 4 5 6 7 8 9

Static local variable:

⇒ If any variable is declaring with the keyword static is called as static variable.

⇒ static variables cannot be reinitialized again and again.

⇒ Static variables will get the memory at "compile time" in Data segment.

⇒ Static local variable also will get the memory in data segment

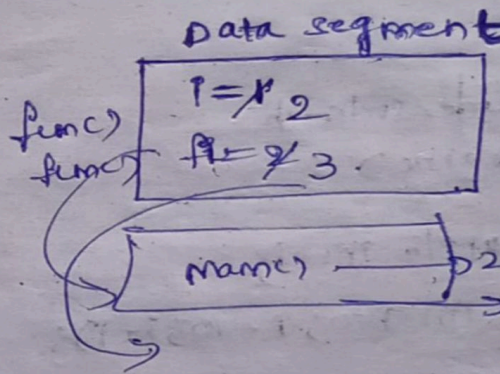
⇒ Static variable have file scope.

⇒ If we cannot initialize any value with static variable, default it takes "0". So, it comes under "uninitialized data segment".

Example:

```
#include<stdio.h>
int func()
{
    static int i=1;
    i++;
    return i;
}
int main()
{
    printf("%d ", func());
    printf("%d ", func());
}
```

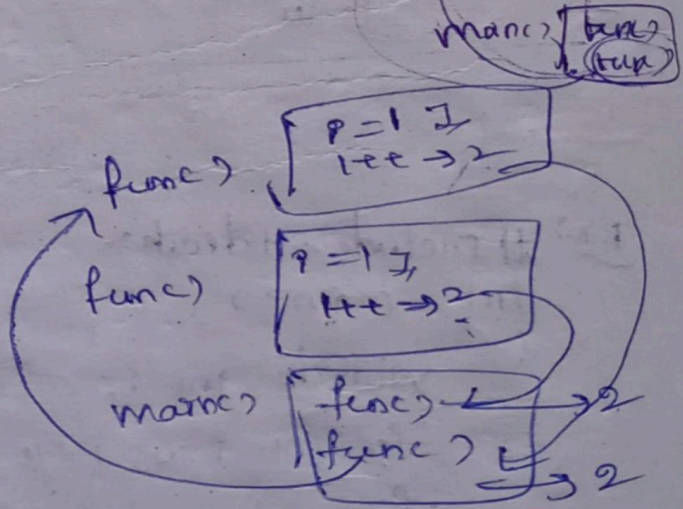
Here if we not put i++, in both cases i may get the result as 1.



O/P: 2 3.

Example:

```
#include<stdio.h>
int func()
{
    int i=1;
    i++;
    return i;
}
int main()
{
    printf("%d ", func());
    printf("%d ", func());
}
```



O/P: 2 2

Memory layout of a C program

⇒ After compiling a C-program one executable file is generated. This executable file have a one memory layout format.

↓

This format is used to run a program in the system.

C program
↓
compile
↓
Executable file → Memory layout format.

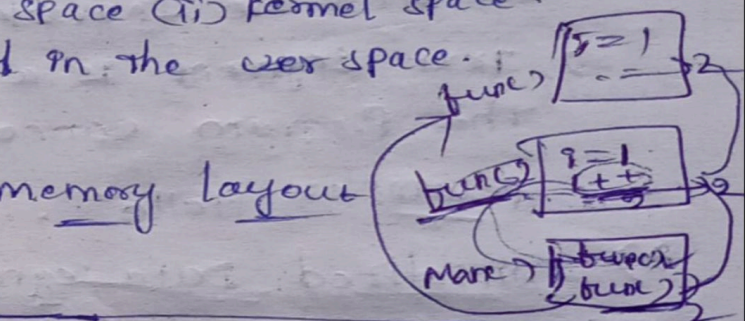
⇒ OS contains two spaces (i) User space (ii) kernel space.

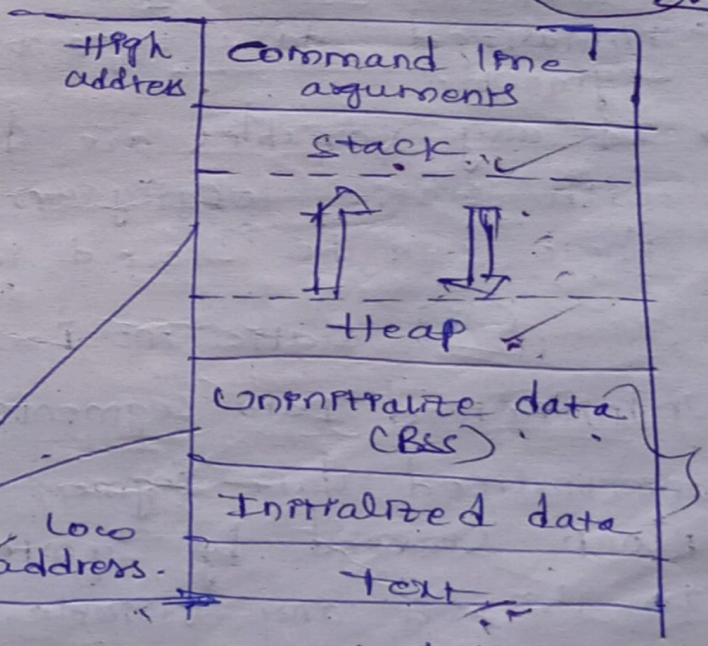
⇒ Applications will be performed in the user space.

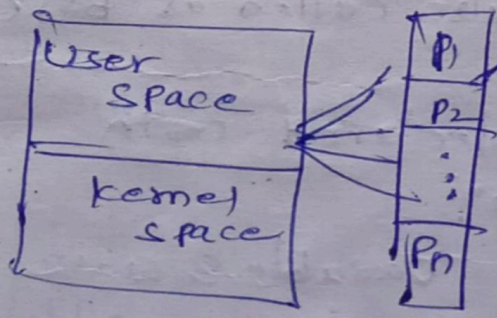
Examples: API, GUI etc.

Memory segments:

1) Text segment.
2) Initialized data segment.
3) Uninitialized data segment (BSS).
   ↓
   Block started by symbol.
4) Stack segment
5) Heap segment







⇒ User space is divided into multiple processors.

⇒ code segment and data segment are created during compile time.

⇒ Stack segment and heap segment are created during run time.

1) Text Segment:

→ This text segment is also called as code segment.

⇒ Whatever the executable instructions in our code, it will be stored in the text segment.

⇒ Generally, this text segment will be placed under the stack segment (or) heap segment.
So, if any stack (or) Heap over may occur, it cannot overlap with the text segment.

⇒ Because of instructions are there in the text segment, so, these are sharable.

⇒ Text segment is a read only memory, so that the Program cannot be changed externally.

Example: "Hello",

2) Initialized Data segments:

→ Initialized data segment usually called data segment.

⇒ If any variables are declared and initialized with static locally (or) globally.
these will be stored in Initialized data segment.

Ex: static int i = 2;

3) Uninitialized Data segments:

→ Uninitialized data segment is also called as BSS (Block Started by symbol).

⇒ In this, if any variable is declared ~~with~~ static by default it takes "0".

⇒ If we can declare a global variable & static variable. with initialized (or) not. these variables will be stored in BSS.

Ex: static int i;
static int i = 0;

4) Stack segment:

⇒ stack memory will be created in the stack segment.
⇒ whenever we call the function, stack frame is created.
⇒ stack follows the LIFO structure.
⇒ Stack frame contains local variable, parameter list and return address.
⇒ return address are compulsory.

> In the stack segment, if we fetch the top element. It is called as stack pointer.

⇒ local variable will get the memory in stack segment depending on the keyword (or) storage class.
⇒ stack area & heap area will grow in opposite direction.
⇒ whenever the stack pointer and heap pointer is intersecting with each other, there is a memory will be free.
⇒ Stack frame will created when the function call encounters, and each stack frame gets deleted after returning the value to the next stack frame.

5) Heap segment:

⇒ Dynamically allocated memory will be stored in the heap segment. like malloc(), calloc() etc.
⇒ this memory will be allocated at a run time.
⇒ libraries are also stored in the heap segment.

> size a.out ↵, which is stored.
> we will get size in each segment

⇒ there is a hole b/w stack segment and heap segment, so, if the stack may overflow heap can access the stack. At this time segmentation fault may occur. Because heap will be at run time, this will not allow to access the stack.

* compilation stages:

when you give gcc filename.c for compiling it generates a.out file.

for this generating we have several steps.

⇒ We have 4 stages to generate a executable file (a.out).

The product of these 4 stages gives a a.out

1) Preprocessing stage.
2) compiling stage.
3) Assembling stage
4) Linking stage

1) Preprocessing stage:

Here all the preprocessive directives gets replaced, header will included to the particular file, comments are removed because whatever the comments we have written is for our understanding, Macros are replaced, conditional compilation also happens.

.c → i/p file. → .i file is generated.

⇒ To see the preprocessor o/p give the command "-E"

gcc -E filename.c → -E → Extended code

-S → (i/o | i/o).

2) Compiling stages

⇒ it takes the i/p as → .i file

⇒ gives the o/p as → .s file.

⇒ the code can be compiled in this stage, the syntax errors will be identified.

gcc -S filename.i.

.i → intermediate file
.c → source file
.s → substitution file

to get the o/p use -S.

3) Assembling stages

⇒ takes the i/p as → .s file.

⇒ gives the o/p as → .o file (object file).

→ to get the o/p use "-c"

gcc -c filename.s

⇒ here, assembly code will converted to Object code.

→ here, errors will not given.

4) Linking: linking of the files will be happen at the linking stage. we may get errors at the linking stage also.

⇒ multiple object files are linked to form one executable file.

.o + .o → a.out

Static Global Variable.

* If any variable is declared with static, it will be stored in data segment.
* If it initialized globally, it will access the throughout program within a file only.
* static have file scope.

Examples: #include<stdio.h>
static int i = 1;
int func()
{
    i++;
    return i;
}
int main()
{
    printf("%d ", func());
    printf("%d ", func());
}

O/p: 2 3

Example for global variables:
#include<stdio.h>
int i = 1;
int func()
{
    i++;
    return i;
}
int main()
{
    printf("%d ", func());
    printf("%d ", func());
}

O/p: 2 3.

⇒ If we declare any variable with globally, it will be stored in "data segment".

Differences b/w Macro and Function.

Macro	Function
1) Macros are preprocessed	1) Functions are compiled.
2) No type checking is done in Macro.	2) Type checking is done in function.
3) Using macros increases the code length.	3) Using function keeps the code length unaffected.
4) Speed of execution using macro is faster.	4) Speed of execution using function is slower.
5) Before compilation, macro name is replaced by macro value.	5) During function call, transfer of control takes place.
6) Macros are useful when small code is repeated many times.	6) Functions are useful when large code is to be written.
7) Macro does not check any compile-time errors.	7) Function checks compile-time errors.

Example for Macro:

```
#Include<stdio.h>
#define NUMBER 10
Int main()
{
    Pfc"%d", NUMBER);
    return 0;
}
```

Olp: 10.

Example for function:

```
#Include<stdio.h>
Int number()
{
    return 10;
}
Int main()
{
    Pfc"%d", number());
    return 0;
}
```

Olp: 10.

⇒ compared to function, Macro is an efficient way to use. Why: The time gets saved in macro. In function, when the function call happens, it will go to definition executes and get back. It consumes some time.

* Differences blw Typedef & Macro:

typedef	Macros
1) typedef is a keyword.	1) Macros are the predefined
2) It is used to create a new name to the existing type.	2) Macros are the symbolic names given to the constants.
3) The compiler performs it.	3) The Pre-processor performs it
4) It uses semicolon to terminate the statement.	4) It does not use a semicolon to terminate a statement.

⇒ typedef can make a complex definition or declaration easier to understand.

⇒ Both typedefs and macros are defined outside the main function.

⇒ typedefs provides to make a program more portable.

Syntax of typedef: typedef <existing_name> <alias_name>

Example:

```
#Include<stdio.h>
typedef unsigned int uint;
int main()
{
    uint i,j;
    i=10;
    j=20;
    Pfc"%d %d", i,j);
}
```

Olp: 10 20.

Example for macro:

```
#include<stdio.h>
#define sum(x,y) (x+y)
int main()
{
    int x,y;
    printf("%d", sum(x,y));
}
```

O/P: 5.

Example for both typedef and macro:

```
#include<stdio.h>
#define int_ptr int *
typedef int * int_ptr;
int main()
{
    int_ptr p1, p2;
    int *p3, p4;
    printf("%zu %zu %zu %zu", sizeof(p1), sizeof(p2),
           sizeof(p3), sizeof(p4));
}
```

Explanation:

int_ptr p1, p2 ⇒
int *p1 ⇒ 8.
int p2 ⇒ 4.
int_ptr p3, p4 ⇒ int *p3 ⇒ 8.
int p4 ⇒ 4.

O/P: 8 4 8 4.

⇒ Compared to both typedefs and macro, typedef is preferable.

Inline function.

→ Inline functions are those function whose definitions are small and be substituted at the place where the function call is happened.

→ Inline function is similar to macro in many ways.

→ Inline is just a request to the compiler. If possible it will inline it. Otherwise it works like a normal function.

→ Inline functions only supports for smaller functions.

In inline:

(i) stacks are not created.
(ii) recursion is not supported.
(iii) loops are not supported.
(iv) It is used to reduce the context switching time.

Example: #include<stdio.h>

```
static inline int sum(int a, int b)
{
    return a+b;
}
int main()
{
    int res = sum(5, 10);
    printf("%d\n", res);
}
```

O/P: 15

It is a gcc specific.
If we won't put it, we will get an error.
In c++, static is not supported.

Differences b/w structure and union.

Structure	Union
1) The keyword "struct" is used to define a structure.	1) The keyword "union" is used to define a union.
2) when a variable is allocated with a structure, the compiler allocates the memory for each member.	2) when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory.
3) The size of the structure is greater than or equal to the sum of size of its members.	3) The size of the union is equal to the size of largest member.
4) structure padding is there in structures.	4) There is no padding in unions.
5) for accessing the structure members, use "." operator and "->" operator.	5) for accessing the union members, use "." operator.
6) #Pragma is used in structures. because to avoid the structure padding.	6) #Pragma is not used in unions.
Structures: 7) It can't allocate common memory for all the members.	7) The common memory is allocated for all the members.

Why structures:

* Under a single name, it can store multiple type of data.
* Structure provides a way to represent a collection of related data items as a single unit.
* Using primitive datatypes, this is not possible.

Example: Student

Student have some important details like id, name, address.
int, char

* Structure Padding:

→ adding useless bytes in between the address of the members is called structure padding.

Why Structure Padding:

to get the proper data alignment and it takes 1 cycle. it depends on the CPU.

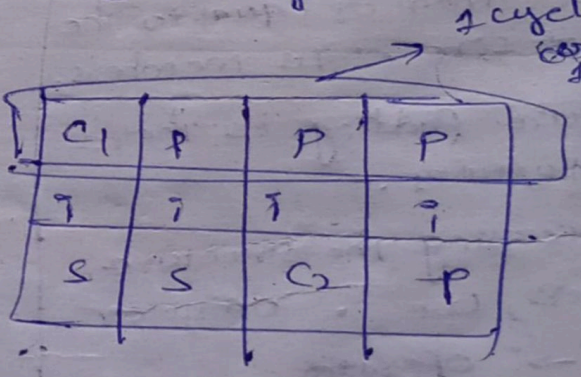
If it is a 32bit, it have 4 bytes.

4 bytes → 1 word (or) 1 cycle.

Example:

```
#include<stdio.h>
struct Student
{
    char c1;
    int i;
    short s;
};
char c2;
int main()
{
    printf("%zu", sizeof(struct Student));
}
```

o/p: 12



Structure size depends on the:

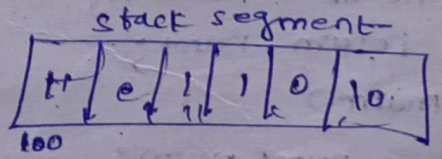
(i) order of the members.
(ii) highest data type.
(iii) size of the structure.

* Differences blw constant strings & modifiable strings:

Constant strings	Modifiable strings
1) Constant strings are often represented as string literals which are sequence of characters enclosed in double quotes. Ex: "Hello, World!" Ex: char *str = "Hello";	1) Modifiable strings are typically represented as character arrays (array of characters) that can be modified. Ex: char str[] = "Hello, World!";
2) String literals are shared in read only memory.	2) The character array can be modified since it is not shared in read-only-memory.
3) These are stored in code (or) text segment.	3) These are stored in stack segment.
4) Constant strings are sharable strings.	4) Modifiable strings are not sharable.
5) Attempting to modify a constant string leads to undefined behaviour (or) segmentation fault. Example: char *str = "Hello"; str[0] = 'h'; // undefined behaviour	5) These strings allowed if we want to modify. Example: char str[] = "Hello"; str[0] = 'h'; // valid modification

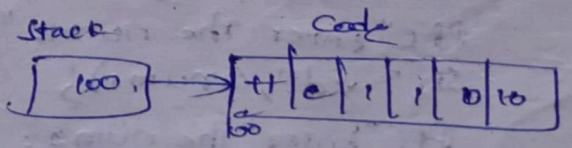
Modifiable string:

char str[] = "Hello";



Constant string:

char *str = "Hello";



* Function: Block of code which performs specific operation and can be called repeatedly.

Why functions:

1) Re usability.
2) Divide & conquer → a big difficult problem divided into smaller.
3) Modularity can be achieved.
4) code can be easily understandable and modifiable.
5) functions are easy to debug and test.
6) abstraction can be achieved in the functions.
   ↓
   hiding the ~~data~~ functionality.

* Properties of functions:

1) takes input
2) perform operation
3) Generates the output

* How can we write program in functions:

→ function declaration / Prototype / Signature.
→ function definition / Body.
→ function call.

functions have two types:

1) In-built functions → Ex: printf, scanf
   ↓
   already defined.
2) User defined functions → which are defined by the user.

1) function declaration:

→ Instructing the compiler about the function.
→ Return type.
→ Number of parameters.
→ Type of parameters.

Syntax: → In function declaration, semicolon should be must

```
return_type function_name (parameter types);
```

Ex: int add (int, int);

return type — function name — parameters/arguments.

Ex: float avg (int, int, float);

Ex: char fun (char);

ex: void fun (int);

ex: void fun1 (void);

Ex: void fun1 ( );

these are the some examples of function declaration.

2) function debinition: → actual logic.

Syntax:

```
return_type function_name (Parameters with type)
{
    // function body/logic.
}
```

Example:

```
int add (int n1, int n2)
{
    int sum = n1+n2;
    return sum;
}
```

// function body / logic.

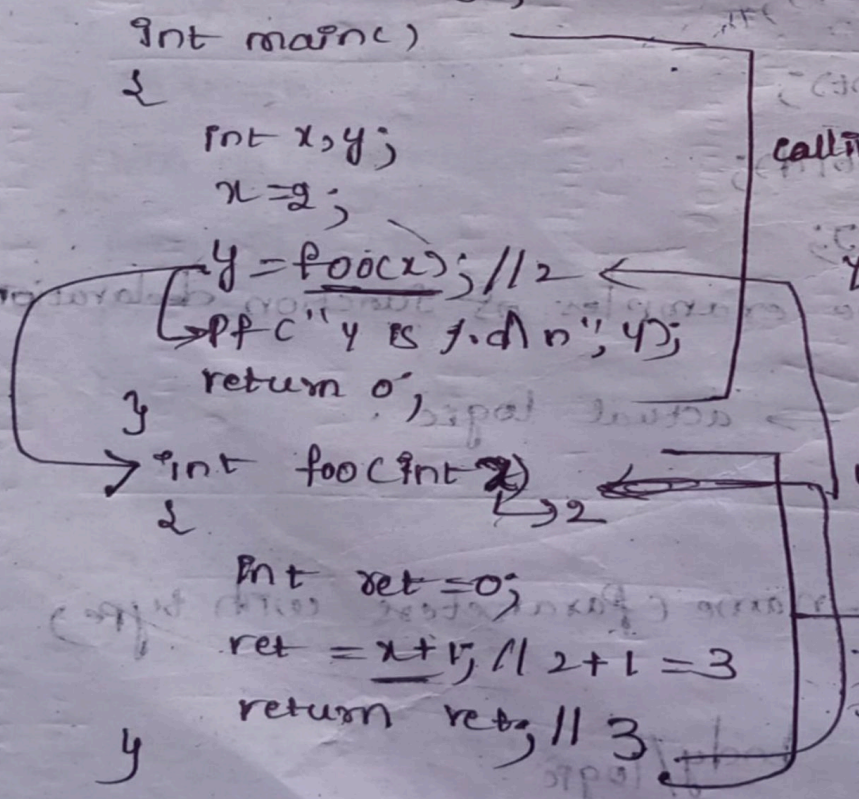
3) function call:

Syntax: `function_name (parameters);`

Example: add (10,20); (or) add (num1, num2);

How to call:

```
#include<stdio.h>
int foo(int x); (or) int foo(int);
int main()
{
    int x,y;
    x=2;
    y=foo(x); //2
    printf("y is %d\n",y);
    return 0;
}
int foo(int x)
{
    int ret=0;
    ret = x+1; // 2+1=3
    return ret; // 3
}
```



→ If we move from one function to another function. Then it is called as "Context switching".

→ Execution is always starts from main function.

Differences b/w Pass by value and Pass by reference.

Pass by value	Pass by reference (return type is "void")
1) In this, we can pass the value through a variable.	1) In this, we can pass the value through the address.
2) more than one value is not modified using pass by value	2) Here, we can modify any number of values by passing through address
3) Here returning is compulsory to see the change in the output.	3) Here returning is not required.
4) It is not possible to return more than one value	4) Here, we return any number of values.
5) It is also called as call by value.	5) It is also called as pass by reference.
6) It can be done using normal functions.	6) It can be done using pointers.

Examples: Pass by values

```
#include<stdio.h>
int pass_value(int num1, int num2)
{
    num1 = num1+1;
    num2 = num2+1;
    return num1;
}
int main()
{
    int num1,num2;
    Pfc("Enter the num1: ");
    Sfc("%d",&num1);
    Pfc("Enter the num2: ");
    Sfc("%d",&num2);
    num1 = pass_value(num1,num2);
    Pfc("%d %d", num1, num2);
    return 0;
}
```

num1: 10
num2: 20

O/P: 11 20

Example for nesting fn in main():

```
#include<stdio.h>
int main()
{
    static int x=0;
    if(x<10)
    {
        x++;
        printf("%d\n", x);
        main();
    }
}
```

O/P: 1 2 3 4 5 6 7 8 9 10

D.S

→ isalnum() ⇒ a-z, A-Z, 0-9 → 8
→ isxdigit() → a-f, A-F, 0 to 9 → 16
⇒ islower() ⇒ a-z → 512
⇒ ispunct() ⇒ spe ch → 4

Example for Pass by reference:

```
#include<stdio.h>
void Pass_ref(int *num1, int *num2)
{
    *num1 = *num1 + 1;
    *num2 = *num2 + 7;
}
int main()
{
    int num1, num2;
    printf("Enter num1: ");
    scanf("%d", &num1);
    printf("Enter num2: ");
    scanf("%d", &num2);
    Pass_ref(&num1, &num2);
    printf("%d %d", num1, num2);
}
```

num1 = 5
num2 = 3+5 = 8

num1: 10
num2: 20

O/P: 4 21

num1 = 5
num2 = 5

* Union Example using ternary operator:

```
#include<stdio.h>
union Endian
{
    unsigned int value;
    unsigned char byte[4];
};
int main()
{
    union Endian e = { 0x12345678 };
    e.byte[0] == 0x78 ? printf("Little\n") : printf("Big\n");
    return 0;
}
```

Little:

78	56	34	12

100

Big:

12	34	56	78

100

MSB LSB

Recursion: The function calling itself is called recursive function.

Ex:

```
int add(int);                    // function definition
{
    add(int);  -> function calling itself
}
```

Here, add is the recursive function

→ Recursion is the process of repeating items in a self-similar way.

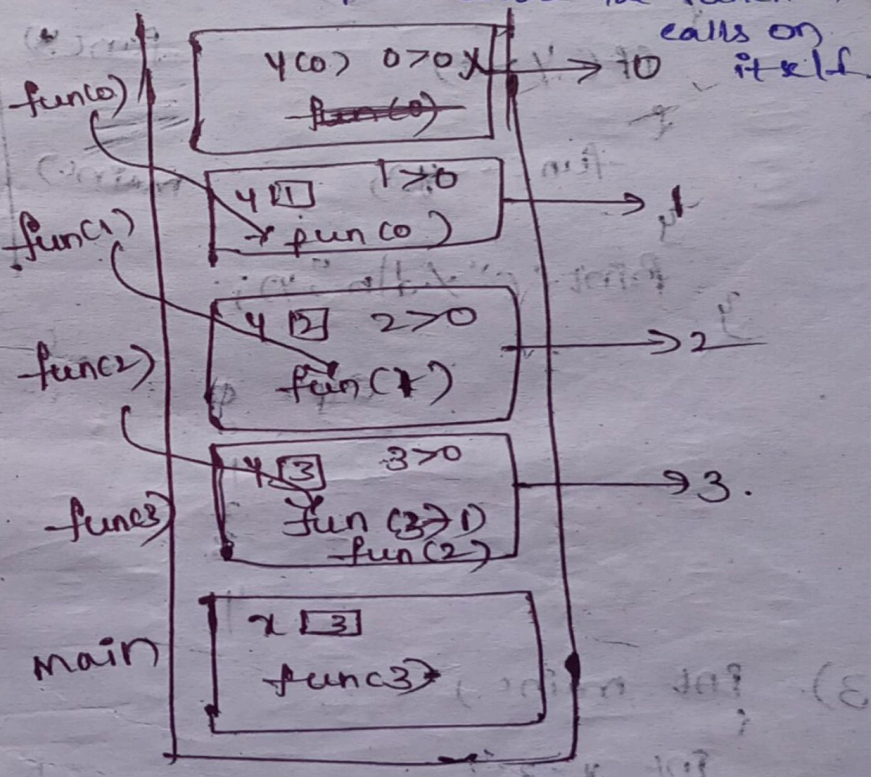
→ two steps is need to write a program using recursion:

step 1: Identification of base case :- The base case condition to stop the recursion.

step 2: writing a recursive case. → The recursive case is the part where the function calls on itself

Example: for recursive function:

```
#include<stdio.h>
int fun(int y);
int main()
{
 int x=3;
 fun(x);
 return 0;
}
int fun(int y)
{
 if(y>0)
 {
  fun(y-1);
 }
  Printf("y.d\n",y);
}
```

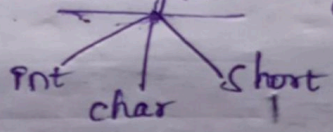


O/P: 0
1
2
3.

* Bitwise Operators:

→ Bitwise operators operates on Bits.

→ Operands should be integral type.



→ Result will be in integer.

There are 6 types of Bitwise operators:

1) Bitwise AND (&)
2) Bitwise OR (|)
3) XOR (^)
4) Complement (~)
5) Left shift (<<)
6) Right shift (>>)

→ N (+ve value) → - increment
Ex: N14 → -15
→ N (-ve value) → + decrement
N -2 → 1

1) Bitwise AND (&):

AND

		o/p
0	0	0
0	1	0
1	0	0
1	1	1

AND	0	1
0	0	0
1	0	1

AND:
If both the i/p's are high. Then the o/p should be high. otherwise low.

→ ~~In the~~ take
We can ~~create~~ the each example in 32 bits.

Ex: 8 & 10.

```
8 :  0000 0000 ..... 1000
10 :  0000 0000 ..... 1010      } do the AND operation
-------------------------------
     0000 0000 ..... 1000  → 8.
```

Ex: -5 & -8

```
-5 :  0000 0000 ..... 0101 → +5
      1111 1111 ..... 1010 → 1's
                          1 → add 1
-------------------------------
      1111 1111 ..... 1011 ⇒ -5
```

```
-8 :  0000 0000 ..... 1000 → +8
      1111 1111 ..... 0111
                          1
-------------------------------
      1111 1111 ..... 1000 → -8.
```

```
-5 :  1111 1111 ..... 1011   } AND
-8 :  1111 1111 ..... 1000
-------------------------------
      1111 1111 ..... 1000 → -8
```

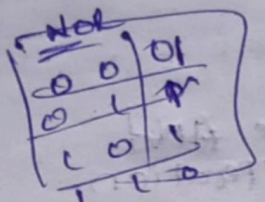
Ex: 249 & 352

249 : 1111 1001

```
2 | 249
2 | 124 - 1
2 | 62 - 0
2 | 31 - 0
2 | 15 - 1
2 | 7 - 1
2 | 3 - 1
    1 - 1
```

2) Bitwise OR (|):

OR:



OR:

0	0	0
0	1	1
1	0	1
1	1	1

If both the i/p's are low. then o/p should be low. Otherwise high.

Ex: 8 | 10. → we can write 8 & 10 values in 32 bits in the form of binary.

8: 0000 0000 1000
10: 0000 0000 ..... 1010 } OR

0000 0000 ..... 1010 ⇒ 10.

3) Bitwise XOR (^):

XOR:

0	0	0
0	1	1
1	0	1
1	1	0

XOR:

If Both i/p's are low and Both i/p's are high. then the o/p should be low.

Ex: 8 ^ 10.

8: 0000 0000 1000
10: 0000 0000 ..... 1010 } do the XOR operation

0000 0000 ..... 0010 ⇒ 2

4) Left shift (<<):

Value << Number of times to be shifted.

Ex: 1 << 2

0000 0000 ..... 0000 0001
0000 0000 ..... 0000 0100 → gets filled with zeroes by

2) $5 << 3 = 40$

0000 0000 .... 0000 0101

0000 0000 .... 0010 1000

= 40

3) $-3 << 4 = -48$

-3: 0000 0000 .... 0000 0011 ⇒ 3

1111 1111 .... 1111 1100

1's

1111 1111 .... 1111 1101 ⇒ -3

1111 1111 .... 1111 1101

1111 1111 1101 0000

adding zeroes because it is a negative value.

-48

2's comple

0000 0000 .... 0010 1111

0000 0000 .... 0011 0000

+48

4) $-13 << 3 = -104$

-13: 0000 0000 .... 0000 1101 ⇒ 13

1111 1111 .... 1111 0010

1111 1111 .... 1111 0011 ⇒ -13

1111 1111 .... 1111 0011

1111 1111 .... 1001 1000

adding zeroes

-104

1's

0000 0000 .... 0110 0111

0000 0000 .... 0110 1000

+104

5) $-9 << 2 = -36$

-9: 0000 0000 ---- 0000 1001

1111 1111 ---- 1111 0110 2's

1111 1111 ---- 1111 0111

1111 1111 ---- 1111 0111

1111 1111 ---- 1101 1100 2's

-36

0000 0000 ---- 0010 0011

+ 1

0010 0100

+36

6) Right shift (>>):

1) Unsigned Right shift

2) Signed Right shift

1) Signed Right shift:

Value >> Number of times to be shifted

Ex: 1) $-1 >> 2$

-1: 1111 1111 ---- 1111 1111

discarded

1 1 1111 1111 ---- 1111 1111

Ex: 2) $4 >> 3$

discarded

0000 0000 ---- 0000 0100

0000 0000 ---- 0000 0000

3) -5 >> 4.

-5: 1111 1111 ..... 1111 1011 → discarded
1111 1111 ..... 1111 1111 ⇒ -1

4) 128 >> 2

0000 0000 ..... 1000 0000 → discarded
⇒ 0000 0000 ..... 0010 0000 ⇒ 32

2) unsigned Right shift:

1) 4294967295 >> 2.

1111 1111 ..... 1111 1111 → discarded
0011 1111 ..... 1111 1111

filled with only zeroes because it is unsigned.

1) for getting "n" bits: that bit should be 1 and remaining all are 0's.

2) clears n bits: that bit should be 0 and remaining all are 1's.

* main()
{
char a,b,c;
a = 135;
b = 89;
c = a+b;
Pf("%d", c);
}

135 - 256 → -121

224 - 256 = -32

o/p:- -32

* Get nth bits:

num & (1<<n)

Ex: num = 10, n = 2

10: 0000 1010
0000 0100 &
0000 0000 ⇒ 0

Result = 0

* Get n bits of a given number (LSB):

num & ((1<<n)-1)

Ex: num = 10, n = 3.

10: 0000 1010 n=3
0000 0111 &
0000 0010 ⇒ 2

Result = 2

1<<3 → 0000 1000 → 8.
8-1 = 7 ⇒ 0000 0111.

* Set n^{th} bits:

num=10, n=2

10: 0000 1010

0000 0100 (1<<2)

0000 1110 → 14.

num|(1<<n)

* Toggle n^{th} bits:

num=10, n=2

10: 0000 1010

0000 0100 (1<<2)

0000 1110 ⇒ 14

num ^ (1<<n)

* clear n^{th} bits:

num=15, n=2

15: 0000 1111

1111 1011 (~(1<<2))

0000 1011 ⇒ 11

Set n bits:

num=10, n=2

10: 0000 1010

0000 0011 (3)

0000 1011 → 11.

num|((1<<n)-1)

Toggle n bits of a number:

num=10, n=2

10: 0000 1010

0000 0011 (3)

0000 1001 → 9.

Clear n bits of a number:

num=15, n=2

15: 0000 1111

1111 1100 (~3)

0000 1100 → 12.

num & ((~(1<<n))-1)

1<<2

0000 0001

0000 0100 → 4

~(1<<2)-1

⇒ 1111 1011

i.e) 4-1=3

00000011

~(3) ⇒ 11111100

Replace n bits:

num = 10
val = 12
n = 3

10 = 00001010 → replace these three bits with 100. which are in 12
12 = 00001100 → clear these bits ie, 00001000

for replacing we use: num & (~((1<<n)-1)). formula.
val

ie, 12: 00001100 n=3

00000111 ~(8)

00000100

1<<3

00001000 ⇒ 8

8-1=7

00000111

or with clear bits and resultant:

00001000 (1)
00000100
00001100 → 12.

resultant 10: 00001100

Advanced functions.

these are of three types:

1) command line Arguments & Environment variables.
2) Variadic functions.
3) function pointers.

1) **Command line Arguments:** Arguments passed from command line is called command line Arguments. These arguments are handled by main function.

Ex:

```
#include<stdio.h>
int main(int argc, char * argv[]);
{
    int a, char *b[]
    return 0;
}
```

syntax

argument vector

having count of arguments

having address

Passed arguments on command line

Arguments count.

argc → argument count → No of arguments to be passed.

argv → argument Vector → denotes the pointer array that is pointing to every argument that has been passed to your program.

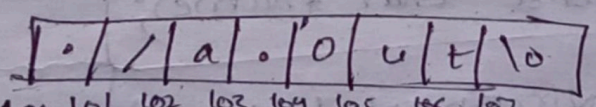
→ ./a.out 10 20

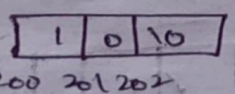
./a.out → 1st argument → argv[0]

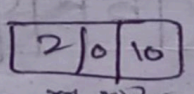
10 → 2nd argument → argv[1]

20 → 3rd argument → argv[2]

Ex

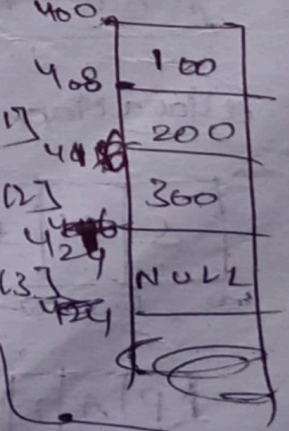


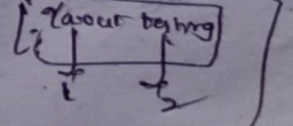





```
Ex: #include<stdio.h>
int main(int argc, char *argv[]);
{
    Pf("Programe name:%s\n", argv[0]);
    if(argc==2)
    {
        Pf("the arg supplied is %s\n", argv[1]);
    }
    else if(argc>2)
    {
        Pf("too many args");
    }
    else
    {
        Pf("No args");
    }
```

argv[0] argv[1] argv[2] argv[3]





Example:

```
#Include<stdio.h>
Int main(int argc, char *argv[])
{
    Int i;
    If(argv[1] != NULL)
    {
        for(int i=0; i<argc; i++)
        {
            Printf("%s ",argv[i]);
        }
    }
}
```

o/p: ./a.out 10 20

./a.out 10 20

Environmental Variables:

→ "envp" is used to ~~get~~ Print all the environmental variables.

Ex:

```
#Include<stdio.h>
int main(int argc, char *argv[], char *envp[])
{
    for(int i=0; envp[i] != NULL; i++)
    {
        Printf("%s\n", envp[i]);
    }
}
```

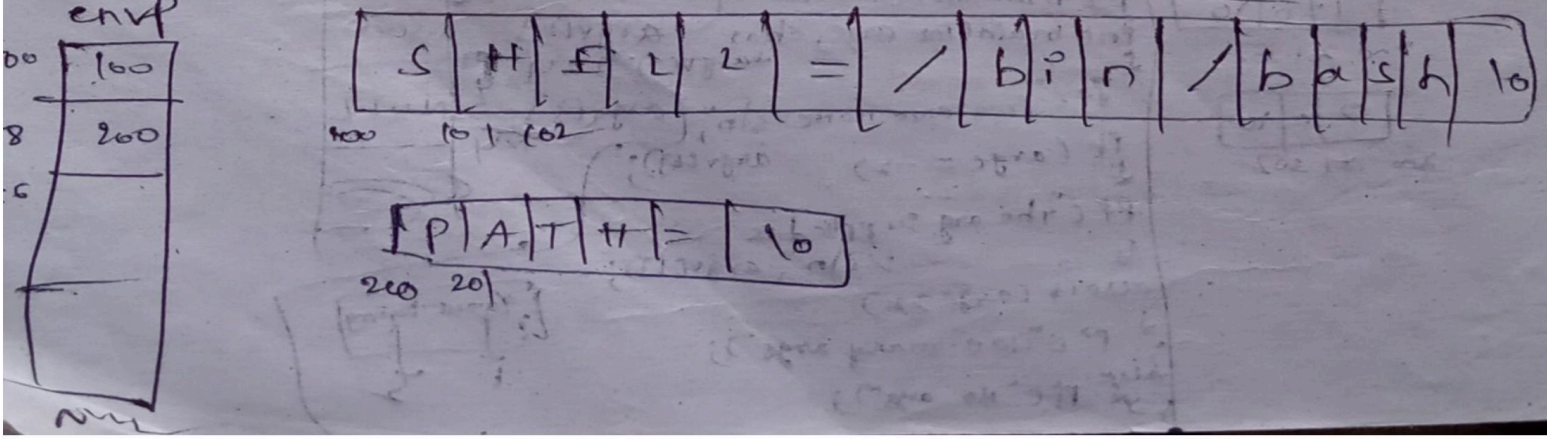
o/p: Print all the environmental variables.

→ GETENV → Used to get an environmental variable.

syntax:

```
char *getenv(const char *name);
```

Memory allocation for Environmental variables:



→ The getenv() function searches the environment list to find the environment variable name, and returns a pointer to the corresponding value string.

Example:

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[], char *envp[])
{
    char *res = getenv("SHELL");
    Printf("%s\n", res);
}
```

Address is 100.

O/P: /bin/bash.

→ If no match is found, it will return success address (or) NULL.

Ex:

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[], char *envp[])
{
    Char *res = getenv("SAI");
    Printf("%s\n", res);
}
```

O/P: Segmentation fault

Ex:

```
Char *res = getenv("AB");
Pf(res != NULL)
    Pf("%s\n", res);
else
    Pf("No match found\n");
```

O/P: No match found.

2) Variadic functions:

→ Variadic functions can be called with any number of trailing arguments.

→ Examples:

printf(), scanf() etc

printf("Hi"); → 1 argument

Pf("%d", 10); → 2 arguments

Pf("%d %d", 10, 20); → 3 arguments

→ arguments will be seperated by comma.

→ Variadic functions can be called in the usual way with individual arguments.

Syntax:

```
return_data_type function_name(parameter list, ...)
```

... represents variadic function.

→ how many successful characters printed on the screen represents printf().

→ Any number of arguments will be passed into the function call.

→ Macros are predefined values.

```
fun(10, 20, 30);
fun(int n, ...)
{
}
```

for 20 & 30 cannot access because, no parameters are there for accesing 20 and 30.

→ So, we can use macros here, to access these 20 & 30.

Argument access Macros:

These macros are defined in the header file "stdarg.h".

1) va_list : Initializes the pointer.
2) va_start: makes pointer to point to the first optional argument.
3) va_arg: It returns the particular argument value & moves pointer to the next argument.
4) va_end: delete (deallocate) the pointer.

Example:

```
#include<stdio.h>
#include<stdarg.h>
int main()
{
    int ret;
    ret = add(3,2,4,4);
    printf("sum is %d\n", ret);
    ret = add(5,3,3,4,5,10);
    printf("sum is %d\n", ret);
    return 0;
}
int add(int count, ...)
{
    va_list ap;
    int iter, sum;
    va_start(ap, count);
    sum = 0;
    for(iter = 0; iter < count; iter++)
    {
        sum += va_arg(ap, int);
    }
    va_end(ap);
}
```

op: sum is 10
sum is 25

for: ret = add(3,2,4,4);
sum = sum + va_arg
sum = 0+2 = 2
sum = 2+4 = 6
sum = 6+4 = 10.

for: ret = add(5,3,3,4,5,10)
sum = 0+3 = 3
sum = 3+3 = 6
sum = 6+4 = 10
sum = 10+5 = 15
sum = 15+10 = 25

→ If the count is not tally, it gives garbage values

Ex: sum(5, 1, 2, 3)
// Count is 5 // but arguments only 3. The next two takes garbage values.

→ sum(3, 1, 2, 3, 4, 5)
// counts // the loop will iterate only 3 times.

fun(3, 1, 2, 3);
// represents count of the total function call // we have count 3 so, these we take three arguments.

Examples:

```
#include<stdio.h>
#include<stdarg.h>
int fun(int, int, ...);
int main()
{
    int ret;
    ret = fun(1, 2, 3, 4);
    return 0;
}
int fun(int n1, int n2, ...)
{
    va_list p;
    va_start(p, n2);
    for(int iter=0; iter<2; iter++)
    {
        printf("%d ", va_arg(p, int));
    }
    va_end(p);
}
```

o/p: 3 4