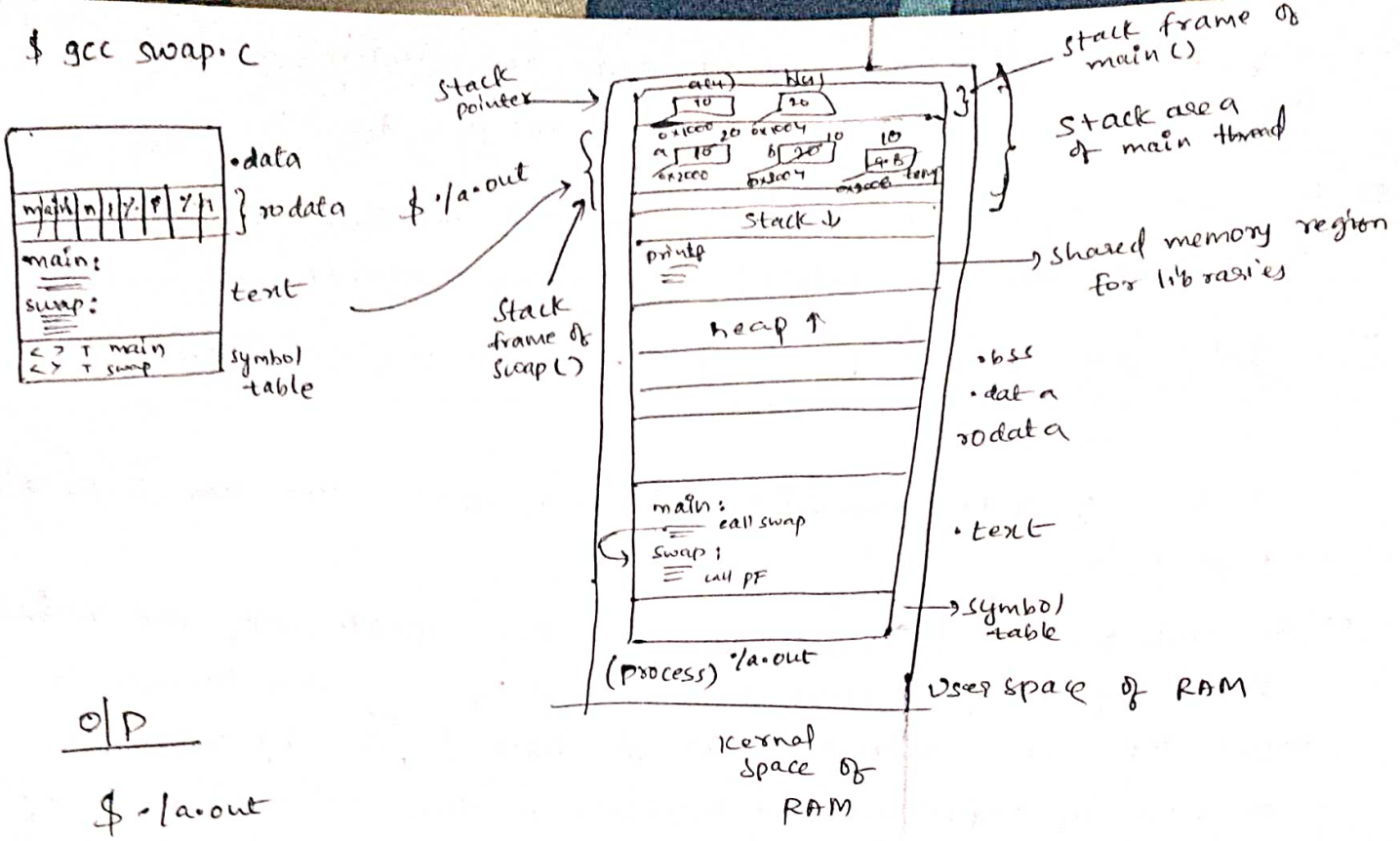
$ gcc swap.c



O/P

$ ./a.out

main : a : 0x1000 – 10
main : b : 0x1004 – 20
swap : a : 0x2000 – 10
swap : b : 0x2004 – 20
swap : a : 0x2000 – 20
swap : b : 0x2004 – 10
main : a : 0x1000 – 10
main : b : 0x1004 – 20

Pointers :-

⇒ pointers are used to access the data with the help of memory location address.

(*) Advantages :-

① To access the data with the help of address.
② To implement call by reference mechanism in functions.
③ To construct array concept (Arithmetic operations on pointers)
④ To access dynamically requested memory.
⑤ Pointer concept + structures concept we can develop Data structures (Arrays comes under Data structures).

⇒ within that process address space if we know the address we can access the data by using pointer variable. Address doesn't have any scope. But, pointer variables will have scope based on their declaration.

⇒ Like, normal variables same life time, scope, default values, storage ~~applications~~ locations are applicable for pointer variables as well.

⇒ when you deal with the pointers, it will come across the runtime errors frequently.

⇒ Pointers must contain valid address memory location.

⇒ Invalid address means the address where we don't have valid data or it can be out of boundary address.

⇒ we can differentiate a pointer variable from a normal variable by dereferencing operator * x (or) * ptr (defaultly use this)

```
main()
{ int x=10;
  ↑
 initialized
 normal local
 variable
```

```
main()
{
  int *x = &y;
    ↑
  initialized
  local integer pointer variable.
```

⇒ * ptr
↑
dereferencing operator / indirectional operator / astric symbol / star.

Ⓐ Steps to be followed to access the data by using pointer variable.

Step:-1 :- Declare (or) define a pointer variable based on data type `int *iptr;`

Step:-2 :- Assign the address of the memory location to the pointer which you want to access `iptr = &x;`

Step:-3 Dereference the pointer to access the data from that address `*ptr = 20;`

⇒ when we dereference the pointer we will get the data present at that address.

Pointer size doesn't depends on data type. It depends upon architecture machine type.

Machine	
32 bit	⟶ 4 bytes
64 bit	⟶ 8 bytes.

⇒ A pointer which contains garbage value (invalid address) it is known as wild pointer.

⇒ A pointer which contains null or zero as the address it is known as Null pointer.

```
ptr = 0 ; (Generally we don't use this)
Ptr = NULL ;
   ⇓
 MACRO
```

⇒ void pointer

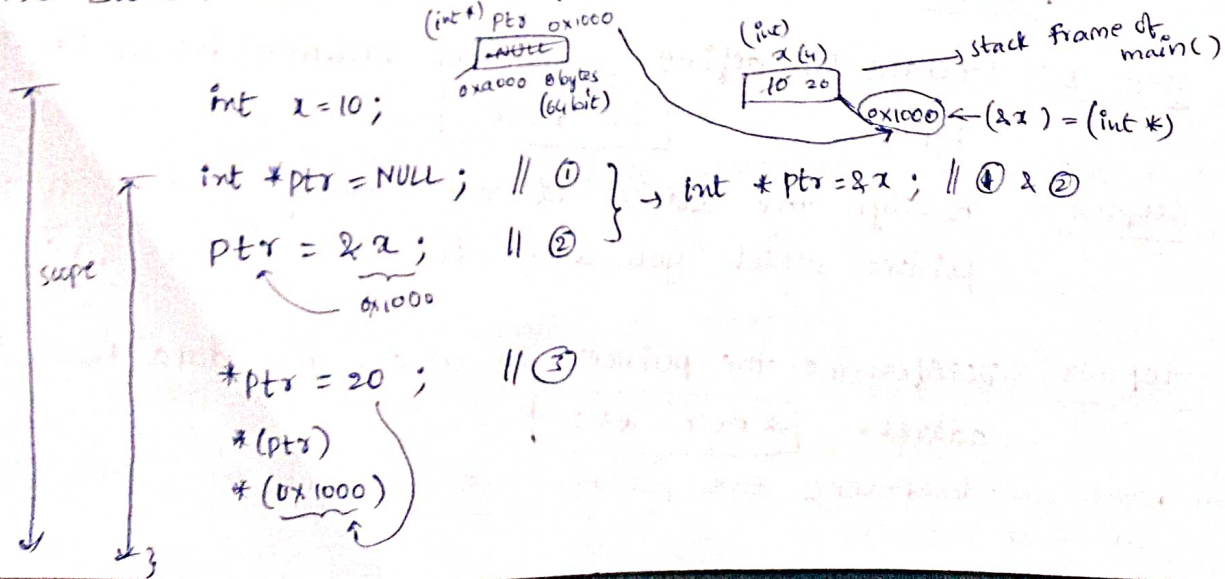
⇒ dangling pointer.

⇒ We must update a pointer with NULL when it is not pointing any valid memory location. This NULL acts as reference.

(like while(x!=0)
while(x>0)
↳ reference point)

⇒ Importance of data type in pointer variable :-

⇒ It signifies or it tells how many bytes to be accessed from the stored address when you dereference the pointer.



Printf ("%p - %d - %d - %d\n", &x, x, sizeof(&x), sizeof(x));

0X1000 20 8 4 (int *) (int)

Printf ("%p - %d\n", ptr, sizeof(ptr));

0X1000 8

Printf ("%d - %d\n", *ptr, sizeof(*ptr));

20 4

Printf ("%p - %d\n", &ptr, sizeof(&ptr));

0X2000 8

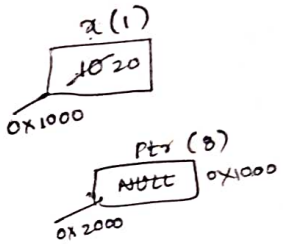
⇒ char *
short int *
float *
double *

⇒ char x = 10;

char *ptr = NULL;

ptr = &x;

*ptr = 20;



PF ("%p - %d - %d - %d\n", &x, x, sizeof(&x), sizeof(x));

0X1000 20 8 2 (int *)

PF ("%p - %d\n", ptr, sizeof(ptr));

0X1000 8

PF ("%d - %d\n", *ptr, sizeof(*ptr));

20 1

PF ("%p - %d\n", &ptr, sizeof(&ptr));

0X2000 8

⇒
```
float x = 10;
float *ptr = NULL;
ptr = &x;
*ptr = 20;
PF("%p - %f - %ld - %ld\n", &x, x, sizeof(&x), sizeof(x));
              0x1000  20.000000  8  4
PF("%p - %ld\n", ptr, sizeof(ptr));
          0x1000  8
PF("%f - %ld\n", *ptr, sizeof(*ptr));
          20  4
PF("%p - %ld\n", &ptr, sizeof(&ptr));
          0x2000
```

⇒
```
double x = 10;
double *ptr = NULL;
ptr = &x;
*ptr = 20;
PF("%p - %lf - %d - %ld\n", &x, x, sizeof(&x), sizeof(x));
          0x1000  20  8  8
PF("%p - %d\n", ptr, sizeof(ptr));
          0x1000  8
PF("%lf - %ld\n", *ptr, sizeof(*ptr));
          20.000000  8
PF("%p - %d\n", &ptr, sizeof(&ptr));
          0x2000  8
```

* NULL - MACRO Definition

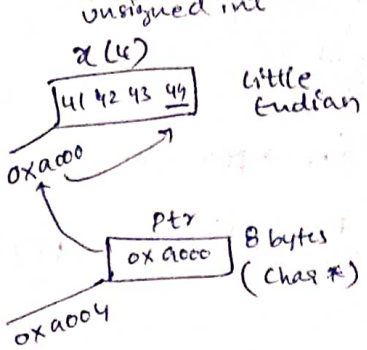
⇒ Used with null-pointer operations & func^{c}.

⇒ Null defined in header files : CRT DBG.H, LOCALE.H, STDDEF.H, STDIO.H, STDLIB, TCHAR.H, TIME.H, WCHAR.H.

⇒ The c library MACRO NULL is the value of a null pointer constant. It may be defined as ((void*)0), 0 or 0L depending on compiler vendor

27/10/22

⇒ unsigned int x = 0X41424344;
unsigned char *ptr = &x;



PF("%x - %d - %c", *ptr, *ptr, *ptr);

↓ 0x44 ↘ 68 ↘ D

(When we pass address of the data it access only 1 byte even if it %x format specifier)

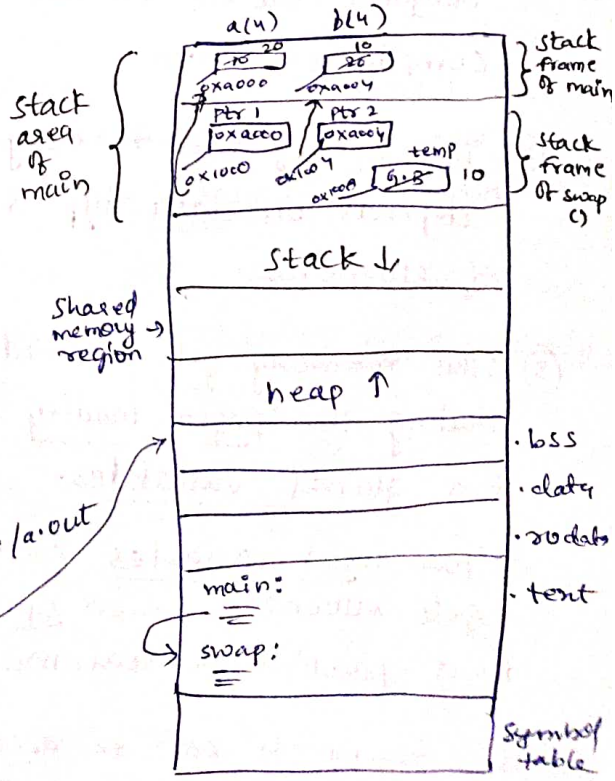
$(44)_{16} = (0100\ 0100)_2 = 64+4 = 68 = 'D'$

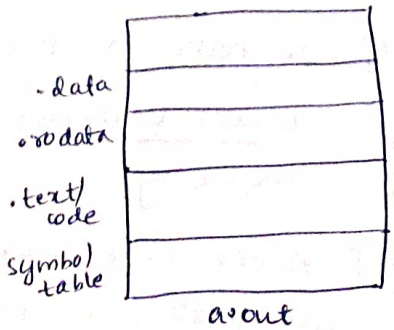
0xa000 0xa004

⇒ void swap (int *ptr1, int *ptr2)

{
int temp;
temp = *ptr1; ⇒ temp = *(0x1000) → 10
*ptr1 = *ptr2; *(0x1000) = *(0x1004) → 20 (indirectly)
*ptr2 = temp; *(0x1008) = temp → 10 (b indirectly)
}

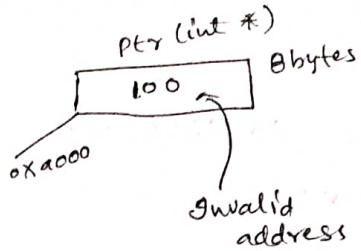
main()
{
int a = 10, b = 20;
swap(&a, &b);
Printf("a-%d \n b-%d \n", a, b);
a=20 b=10
}



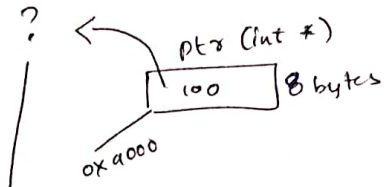


./a.out

⇒ main()
{
int *ptr = 100;
printf("%d\n", ptr);
}
↳ 100



⇒ main()
{
int *ptr = 100;
PF("%d\n", *ptr);
}



↳ segmentation fault error.

⇒ Dynamic Memory allocation calls (DMA calls)

⇒ Differences b/w statically requested memory & dynamically requested memory :-

⇒ Statically requested memory	Dynamically requested memory
① Memory is decided by the compiler based on data type at compilation time	① This memory can be requested dynamically by using DMA calls. i.e., calloc(), malloc().
② The size of this memory depends on data type & no. of elements.	② This memory gets allocated in heap segment.
③ This memory gets allocated during the pgm loading time for global variables. ⇒ For local variables memory gets allocated when CPU executes that particular statement.	③ This memory can be accessed by the address returned by DMA calls.
④ This memory can be accessed by using variable name & also with its address (variable's base address).	④ we have to use dedicated pointer variables to access that memory.
	⑤ After performing the task, we can deallocate that memory by using free fun°.

⑤ This memory gets deallocated

* for local variables → when Fun^c (or) block completes.

* for global variables → when pgm terminates.

⑥ These variables will have scope. They are valid within the fun^c or file scope or block scope.

⑦ → Local variables memory gets allocated in stack frame of corresponding fun^c.

⇒ Global variables memory gets allocated in bss or data segment

⑧ Statically requested memory, neither we can increase or decrease nor dellocate at run time.

⑨ Here, we can't utilize this memory effectively & efficiently.

⑤ This memory size can be increased or decreased by using realloc ( ) during runtime.

⑥ This dynamically requested memory doesn't depends upon data type. We can store any type of data in this memory based on our requirement.

⑦ If we know the address of this memory we can access this from any function.

⑧ Dynamically requested memory doesn't have any scope. We can access them within that address space.

⑨ Improper handling of this memory may leads to memory leaks.

⇒ The developer has to take care of these memory leaks. To findout these memory leaks, valgrind tool is used

⑩ After deallocating the memory by using free ( ), the pointer contains the address where we don't have valid memory, that pointer is known as Dangling pointer

⑪ we can utilize the memory here effectively & efficiently by dynamically requested memory.

⇒ Differences between calloc() & malloc()

① → malloc() will allocate the memory as a single block
→ calloc() will " " " block by block continuously (or) 'n' no. of blocks.

② → malloc() will take the single i/p i.e., overall size of the block
→ calloc() will take 2 inputs i.e., No. of blocks & block size.

③ → malloc() will allocate the memory as it is with garbage values.
→ calloc() will " " " by updating with zeroe's (0).

④ → malloc() is faster than calloc().
→ calloc() will internally invokes ~~two~~ functions (memset functions).

⇒ These malloc(), calloc(), realloc() these library functions internally invokes brk() or sbrk() system calls.

⇒ These DMA calls we have to include stdlib.h header file.
⇒ void pointer is known as generic pointer. eg:- (void *ptr;)

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

(size_t: ↑ typedef of int type)

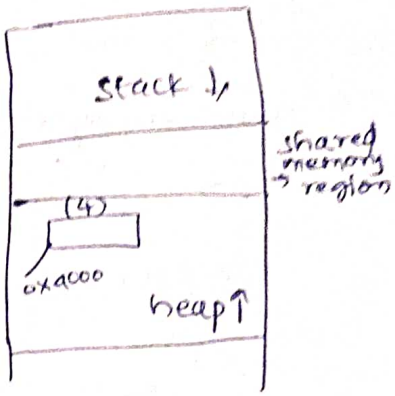
⇒ This malloc() will take the single input i.e., no. of bytes as a size.
⇒ The memory requested by malloc() gets allocated in heap segment & it will return the starting base address.

⇒ malloc() on failure in allocating the memory it will return null. Dereferencing null we will get segmentation error

integer const → 4 bytes

0xa000

```
int *ptr = malloc(4);
// heap → full
// malloc() will return → NULL
```



⇒ when heap segment is full, (if) when we are requesting the memory by using malloc(), calloc(), realloc() these functions will return null on failure

Verify the return values of these DMA calls. if it

⇒ If it is NULL, terminate the application by calling exit() to avoid runtime errors.

```
if(ptr == NULL)
  {
    PF("Fail to allocate memory in heap segment\n");
    exit(1);
  }
```

→ Non-zero value.

syntax for malloc()

DMA CALLS

returns add 0xa000

For integer constant = (4 bytes) 10

```
int *ptr = malloc(4);
```

(It is not a standard way of writing the declaration statement)

```
int *ptr;
ptr = (int *) malloc(1 * sizeof(int));
```

0xa000

(standard type) (or) way

```
if (ptr == NULL)
  {
    PF("fail to allocate memory in heap seg\n");
    exit(1);
  }
  PF("%p - %d\n", ptr, *ptr);
     0xa000   10
  free(ptr); // mandatory
```

** To deallocate the memory after the execution.

```
}
```

```
Struct employee
    {

    }
Struct employee *ptr;
Ptr = (struct employee *) malloc (1 * sizeof (struct employee));
if (ptr == NULL)
  {
    PF(" ");
    exit(1);
  }
```

Syntax for Free ():-

```
void free (void *ptr);
```

⇒ By using this free function, we can deallocate the memory which is requested dynamically (malloc(), calloc(), realloc()).

⇒ This fun^c will take single input i.e., the address returned by malloc or calloc or realloc functions.

⇒ This fun^c will not return anything.

⇒ After deallocating the memory, the pointer contains the address where the memory does not belongs to process address space. That pointer is known as Dangling pointer. We should not dereference the dangling pointer. (don't access the memory from that address in the process).

⇒ Dangling pointers contains invalid address.

28/10/22

⇒ Calloc ( ) :-

⇒ This function will allocate the memory at run time in heap segment as a continuous blocks and returns the starting base address that memory is initialized with zero's.

(*) Declaration of calloc ( )

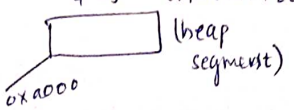
```
void * calloc (size_t nmem, size_t size);
```

```
int * ptr = (int *) calloc (1, size of (int));
```

0xa000

1 → No. of elements

size of (int) → Each element size.



⇒ main( )

```
{
  int x=10;
  int *ptr = (int *) malloc (1 * size of (int));
  int * ptr = (int *) calloc (1, size of (int));
```

0xa000

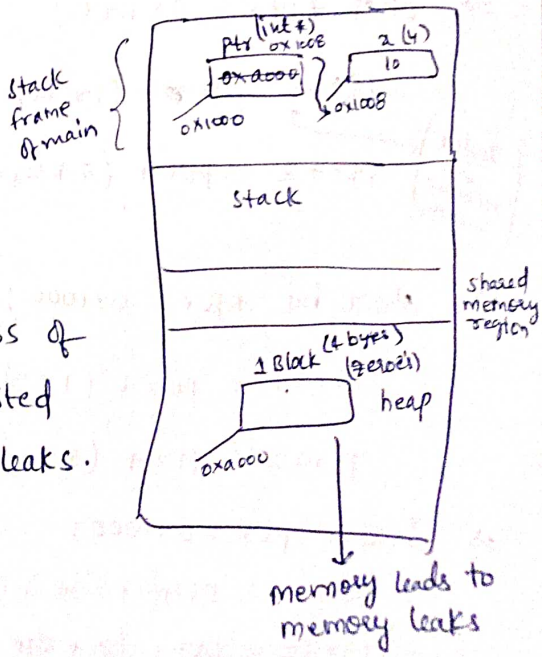
1 → no. of elements

size of (int) → Each element size

```
  if (ptr == NULL)
  {
    PF(" failed to allocate memory \n");
    exit(1);
  }
// ptr = &x;
```

0x1008

```
  *ptr = 10;
  PF(" %p - %d\n", ptr, *ptr);
}
free (ptr);
```



(*) NOTE :-

As soon as we update base address of variable x, the dynamically requested memory may leaks to memory leaks. (0xa000) till the prgm terminates.

⇒ The 4 bytes memory at address 0x a000 cannot be accessed by current process or by other processes till process gets terminated.

⇒ we must use dedicated pointers for dynamically requested memory throughout the process until we deallocate that memory on process termination.

* Arithmetic operations on pointers :-

⇒ If we have 'n' bytes of data, i want to access the data byte by byte we have to use character pointer (char *ptr) with arithmetic operations

⇒ If we want to access same 'n' bytes (data or) memory → 2 bytes, 2bytes we have to use short int pointer with arithmetic operations.

⇒ If we want to access same 'n' bytes of data 4 bytes 4bytes from a given address, we have to use integer pointer with arithmetic operations.

⇒ when we increment character pointer it will jump 1 byte
⇒ when we increment short int pointer " " " 2 bytes.
⇒ " " " integer pointer " " " 4 bytes.

⇒ char *ptr = 0x1000;

ptr+1 = ptr + (1 * size of (char)) = 0x1000 + (1*1) => 0x1001

ptr+2 = ptr + (2 * size of (char)) = 0x1000 + (2x1) ⇒ 0x1002

(Index or reference values)

⇒ short int *ptr = 0x1000;

ptr+1 = ptr + (1 * size of (short int)) = 0x1000 + (1*2) ⇒ 0x1002

ptr+2 = ptr + (2 * size of (short int)) = 0x1000 + (2*2) ⇒ 0x1004

⇒ int * ptr = 0x1000;

ptr+1 = ptr + (1 * size of (int)) = 0x1000 + (1 * 4) => 0x1004

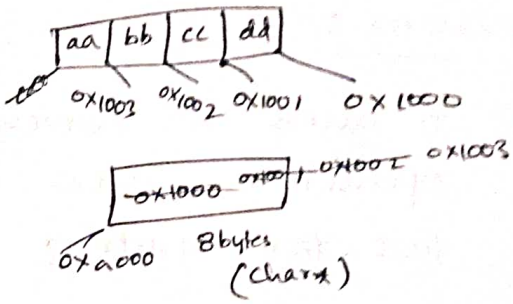
ptr+2 = ptr + (2 * size of (int)) = 0x1000 + (2*4) ⇒ 0x1008

Ex:-

⇒ If we have 2000 bytes memory, we have stored 500 Elements information, If we want to access 100th Element info we have to write `ptr+(100-1)`

⇒ unsigned int x = 0xaabbccdd;

unsigned char *ptr = &c;




```
Printf("%p - %x\n", ptr, *ptr);
        0x1000 - dd
ptr++; // ptr = ptr+1;
Printf("%p - %x\n", ptr, *ptr);
        0x1001   bb
ptr++;
printf("%p - %x\n", ptr, *ptr);
        0x1002  bb
ptr++;
Printf("%p - %x\n", ptr, *ptr);
        0x1003   aa
```

⇒ This method is not recommended

Instead use ⇒ 4 methods

① ptr[i]

② i[ptr]

③ *(ptr+i)

④ *(i+ptr)

(ptr+0), → *(ptr+0)

(ptr+1), → *(ptr+1)

(ptr+2), *(ptr+2)

(ptr+3), *(ptr+3)

```
char *ptr;
  ptr = 0x1000;
PF("%p\n", ptr++);   → 0x1001
  ptr = 0x1000;
PF("%p\n", ++ptr);   ⇒ 0x1001
```

⇒ In arithmetic operation of pointers the first method is not applicable recommendable.

*ptr
ptr++
} Not recommendable

Reasons :-

① To access n^{th} element, we have to perform (n-1) arithmetic operations. After performing those many operations we lost the starting base address.

② we can access the elements only in forward direction.

③ This method will degrade the performance of CPU. There is is a possibility of memory leaks when you deal with dynamic memory.

④ Reaccessing the elements from the begging is not possible in this method.

⇒ Apart from this method we can follow other 4 methods.

To access data/elements

2nd method = *(ptr+i)

III " = *(i+ptr)

accessing elements by using array method

IV " = ptr[i]

V " = i[ptr]

* square braces represents dereferencing

→ *(ptr+i)

→ = *(i+ptr)

To access address

I = ptr

II - (ptr+i)

III - &ptr[i] ⇒ &(*(ptr+i))

IV - &i[ptr] ⇒ &(*(i+ptr))

5 integers : 10, 20, 30, 40, 50 (20 bytes)

```
main()
{
  int *ptr;
  int i;
// ptr = (int *) malloc (5 * sizeof(int));
  ptr = (int *) calloc (5, sizeof(int));
   if (ptr==NULL)
    {
      printf ("failed to allocate memory in heap\n");
      exit (1);
    }
ptr[0] = 10;
ptr[1] = 20;
ptr[2] = 30;
ptr[3] = 40;
ptr[4] = 50;
```

(or)

```
scanf("%d", &ptr[0]);
scanf("%d", &ptr[1]);
scanf("%d", &ptr[2]);
scanf("%d", &ptr[3]);
scanf("%d", &ptr[4]);
```

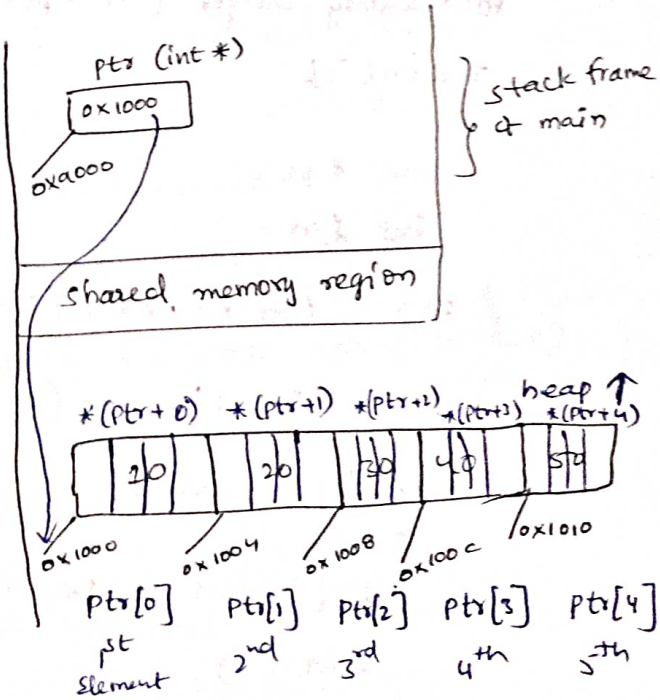
ptr[0] = *(ptr + (0 * sizeof(int)))
= *(0x1000 + 0) ⇒ *(0x1000) — 4 bytes

ptr[1] = *(ptr + (1 * sizeof(int)))
*(0x1000 + 4) ⇒ *(0x1004) — 4 bytes

ptr[2] = *(ptr + (2 * sizeof(int)))
*(0x1000 + 8) ⇒ *(0x1008) — 4 bytes

ptr[3] = *(ptr + (3 * sizeof(int)))
*(0x1000 + (3*4)) ⇒ *(0x100C) — 4 bytes
12 = 0xC

ptr[4] = *(ptr + (4 * sizeof(int)))
*(0x1000 + (4*4)) ⇒ *(0x1010) — 4 bytes
16 = 0x10



```
for (i=0 ; i<5 ; i++)        <-- 5 elements
 {
   printf ("enter the i/p \n");
   scanf ("%d", &ptr[i]);
 }

 for (i=0 ; i<5 ; i++)
```

II nd ⇒ `PF("%d's element → %p - %d\n", i+1, ptr+i, *(ptr+i));`

III ⇒ `PF("%p - %d\n", i+ptr, *(i+ptr));`

IV ⇒ `PF("%p - %d\n", &ptr[i], ptr[i]);`

I ⇒ `PF("%p - %d\n", &i[ptr], i[ptr]);`

```
    free (ptr);
 }
```

Using functions

```
#define MAX 5
int read_integer (int *ptr, int n);
Void display_integer (int *ptr, int n);
main()
 {
   int *ptr;
   int i,n;
// ptr = (int *) malloc (5 * sizeof(int));
   ptr = (int *)calloc (5, sizeof(int));
   if (ptr==NULL)
   {
     PF("failed to allocate memory in heap\n");
     exit (1);
   }
n = read_integer (ptr, 5);      (ptr, MAX)
   display_integer (ptr, n);    (ptr, MAX)
   free (ptr);
}
```

20 bytes address (5 elements)

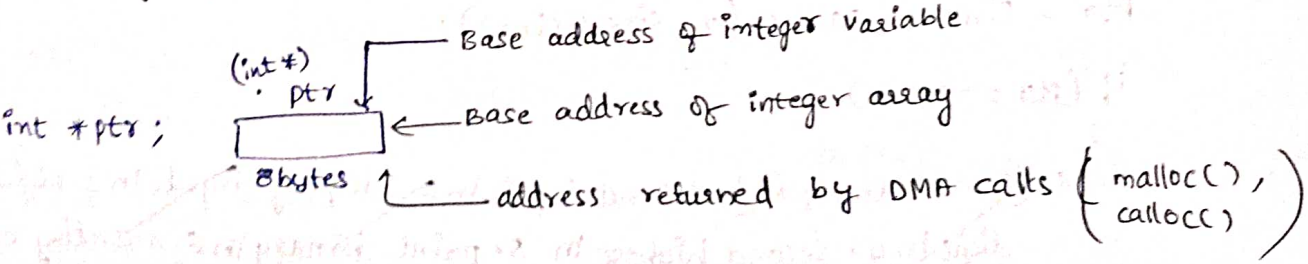
ptr → 0x1000

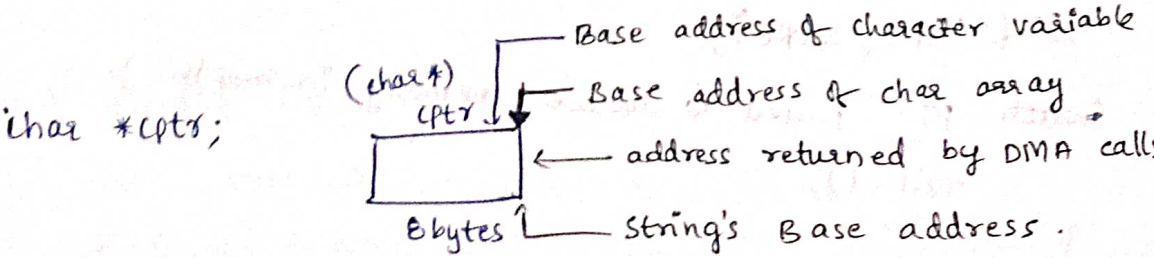
No. of elements

```
int read_integer (int *ptr, int n)
{
  for (i=0 ; i<n ; i++)
  {
    PF("Enter the i/p\n");
    SF("%d", &ptr[i]);
  }
  return i;
}

void display_integer (int *ptr, int n)
{
  for (i=0, i<n ; i++)
  {
  I  PF("%d's Element → %p - %d\n", i+1, ptr+i, *(ptr+i));
  II PF("%p - %d\n", i+ptr, *(i+ptr));
  III PF("%p - %d\n", &ptr[i], ptr[i]);
  IV PF("%p - %d\n", &i[ptr], i[ptr]);
  }
}
```

0x1000 (ptr) — 5 (n)

* A variable address can be stored in a single pointer.
* A single pointer base address can be stored in a double pointer





29/10/2022

Task

0. exit
1. read i/p's
2. display i/ps
3. biggest value
4. 2nd biggest value
5. Binary
6. ascending order

⇒
```
#include <stdio.h>
#include <stdlib.h>
int read_inputs (int *ptr, int n);
int display_inputs (int *ptr, int n);
int highest_digit(int *ptr, int n);
int second_highest (int *ptr, int n);
int ascending_order (int *ptr, int n);
int Print_binary (int *ptr, int n);
int main()
 {
   unsigned int n, *ptr, opt;
   PF("enter n value (no. of elements)\n");
   SF("%d", &n);
  Ptr = (int *) calloc (n, sizeof(int));
  if (Ptr == NULL)
   {
     PF("MENU \n0. exit \n1. read inputs \n2. display inputs \n3. highest
       digit \n4. second highest \n5. print binary \n6. ascending order\n");
     SF("%d", &opt);
   switch  PF("failed to allocate memory in heap\n");
           exit(1);
   }
while (1)
{
   -- fpurge(stdin);
   PF("menu\n 0. exit\n 1. read inputs\n 2. display inputs\n 3. highest
element digit\n 4. second highest\n 5. print binary\n 6. ascending order\n");
```

```
scanf("%d", &opt);
switch(opt)
{
  case 0: exit(0);
  case 1: printf("Enter n values \n");
          read_inputs(ptr, n);
          break;
case 2: PF("displaying the i/p's \n");
        display_inputs(ptr, n);
        break;
                                  case 3: highest_digit(ptr, n);
                                          break;
                                  case 4: second_highest(ptr, n);
                                          break;
                                  case 5: print_binary(ptr, n);
                                          break;
                                  case 6: ascending_order(ptr, n);
                                          break;
                                  default: printf("Invalid option\n");
  }
}
  free(ptr);
}
```

⇒
```
int read_inputs(int *ptr, int n)
{
  int i;
  PF("Enter the i/p's \n");
  for(i=0; i<n; i++)
  SF("%d", &ptr[i]);
  return 0;
}
```

⇒
```
int display_inputs(int *ptr, int n)
{
  int i;
  for(i=0; i<n; i++)
  {
    PF("%p-%d\n", &ptr[i], ptr[i]);
  }
  return 0;
}
```

⇒
```
int highest_digit(int * ptr, int n)
{
  int i, max = 0;
  for(i=0; i<n; i++)
  {
    if(max < ptr[i])
      max = ptr[i];
  }
  PF("highest digit is: %d\n", max);
  return 0;
}
```

④ ⇒
```
int second_highest (int *ptr, int n)
{
    int big=0, i=0, _2big=0;
    for(i=0; i<n; i++)
    {
        if (ptr[i]>big)
        {
            _2big = big;
            big = ptr[i];
        }
        Else if (ptr[i] > _2big)
            _2big = ptr[i];
    }
    PF("second highest = %d\n", _2big);
}
```

⑤
```
int print_binary (int *ptr, int n)
{
    int i, j=1, rem=0, sum=0, temp;
    for(i=0; i<n; i++)
    {
        temp = ptr[i];
        sum=0;
        rem=0;
        j=1;
        while (temp>0)
        {
            rem= temp % 2;
            temp = temp /2;
            sum= sum + rem*j;
            j = j*10;
        }
        PF ("%d-%d\n", ptr[i], sum);
    }
}
```

⑥
```
int ascending_order (int *ptr, int n)
{
    int i, j, temp;
    for (i=0; i<n; i++)
    {
        for (j=i+1; j<n; j++)
        {
            if (ptr[i] > ptr[j])
            {
                temp = ptr[i];
                ptr[i] = ptr[j];
                ptr[j] = temp;
            }
        }
        PF("%d\n", ptr[i]);
    }
}
```

31/10/2022

→ on success it will return same add
→ on failure it will return NULL

⑧ Realloc ( ) :-

starting address of memory requested by malloc(), calloc()

```
void * realloc ( void *ptr , size_t size );
                                    → new overall size.
```

⇒ This function is used to increase or decrease the size of dynamically requested memory by using malloc, calloc fun^{cs} at same address with new size.

⇒ For realloc ( ) also we have to use a separate pointer variable to avoid memory leaks.

5 integers — 10, 20, 30, 40, 50 / 60, 70, 80, 90, 100

To increase the memory size using realloc ( )

```
int *ptr , *nptr ;
ptr = (int *) malloc (5 * sizeof (int));   // 0xa000
   if (ptr==NULL)
     {
       PF(" failed to allocate memory in heap segment \n");
       exit (1);
     }
```

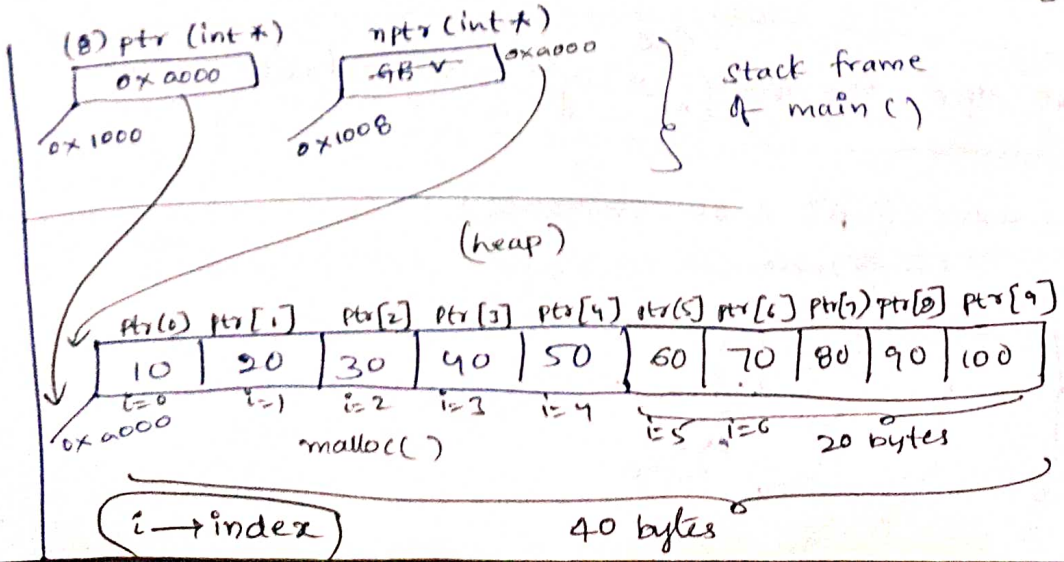
```
ptr [0] = 10 ;
ptr [1] = 20 ;
ptr [2] = 30 ;
ptr [3] = 40 ;
ptr [4] = 50 ;
```

To scan i/ps manually or previously

(or)

```
for (i=0 ; i<5 ; i++)
 {
  scanf (" %d", &ptr[i]);
 }
```

To give i/ps during run time.



0xa000

```
nptr = (int *) realloc ( ptr , 10 * sizeof(int));
                         0xa000
if (nptr == NULL)
 {
   PF("Failed to increase the memory in heap\n");
   exit (2);
 }
```

```
Ptr [5] = 60;
Ptr [6] = 70;
Ptr [7] = 80;
Ptr [8] = 90;
Ptr [9] = 100;
```

or

```
for (i=5; i<10; i++)
 {
 scanf ("%d ", &ptr[i]);
 }
```

```
for (i=0; i<10; i++)
 {
   PF("%p - %d\n", &ptr[i] , ptr[i]) ;
 }                 &nptr[i]   nptr[i]
```

```
free (ptr);   // free (nptr);
      ↓
   0xa000
```

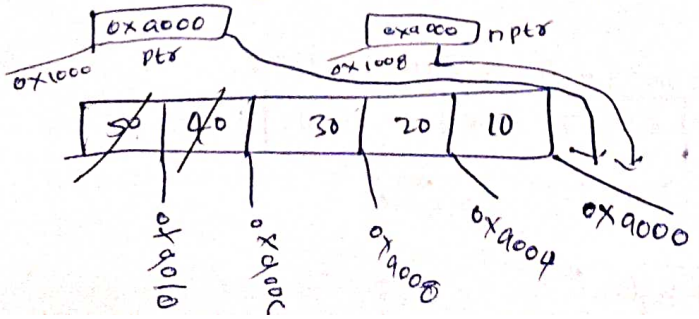
⇒ It will deallocate overall 40 bytes memory

⇒ Now, ptr, nptr consists of same address even after deallocating that memory, then this pointers are known as dangling pointer

⇒ To deallocate the memory we can use realloc with 0 without using free() & prgm termination }.

To decrease the memory size using realloc ( )

Decrease 8 bytes (50 & 40 elements)



```
nptr = (int *)realloc( ptr , 3 * sizeof(int));
       0xa000          0xa000   12 bytes

if (nptr == NULL)
  {
    printf(" failed to decalase the memory \n");
    exit (2);
  }
for (i=0; i<3; i++)
  printf(" %p - %d \n", &nptr[i], nptr[i]);
free (nptr);
```

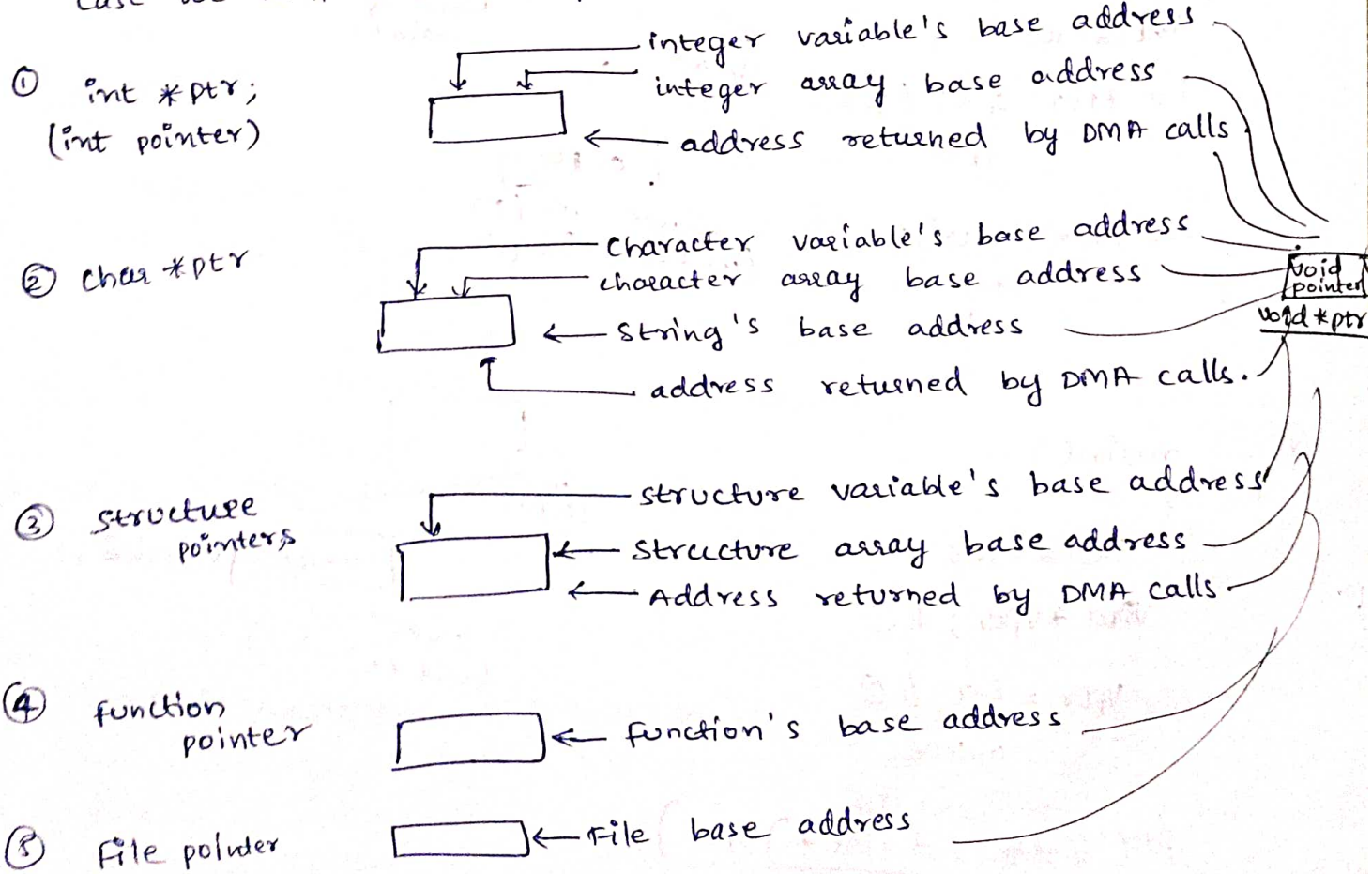
↳ It will deallocate 12 bytes of memory

Now, nptr & ptr ⇒ dangling pointers

* Void pointers :-

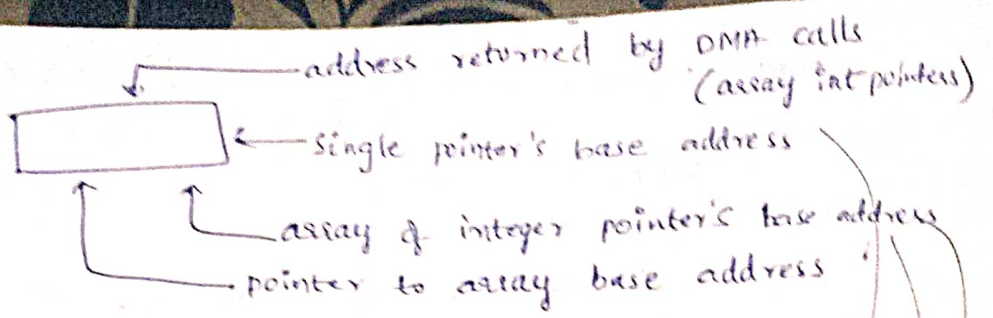
⇒ void pointer is also known as genric pointer.

⇒ If we don't know the type of data to be accessed, in this case we will use void pointer.



⑥ int **dptr ;

double pointer



⑦ char **dptr

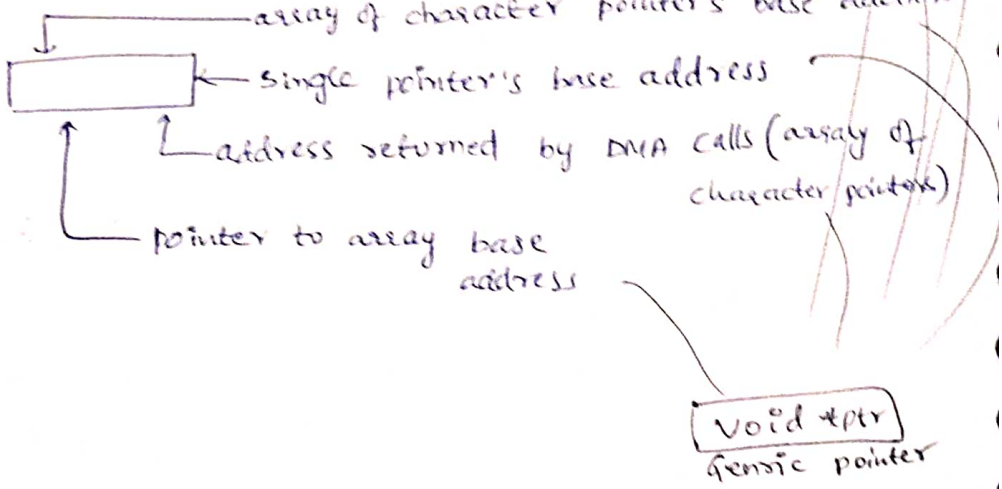
(character double pointer)

|

|

|

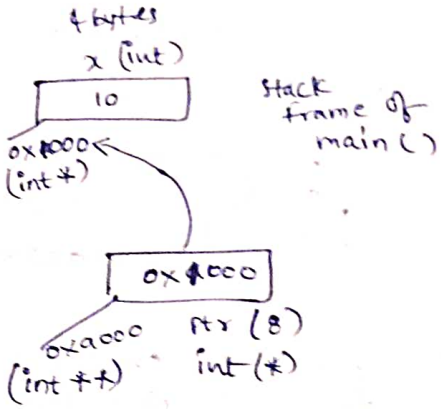
etc



⇒ To access the data by using void pointer we have to type cast the void pointer based on data

⑧

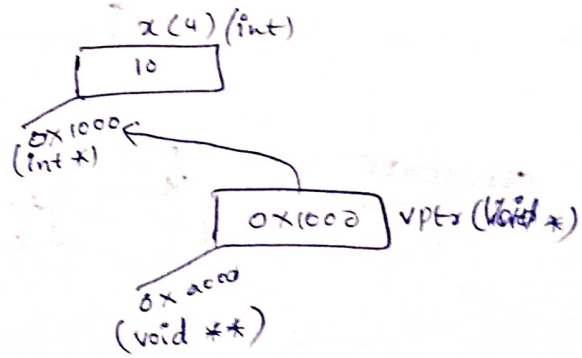
```
main()
{
  int x=10;
  int *ptr = &x; // ① & ②
                 0x1000
  PF("%d\n", *ptr);
             *(0x1000)
                10
}
```



⑨

```
main()
{
  int x=10;
  void *vptr; // ①
  vptr = &x; // ②
             0x1000
  PF("%d\n", *(int*)vptr);
}
```

Type casted void pointer to int type to access the data.

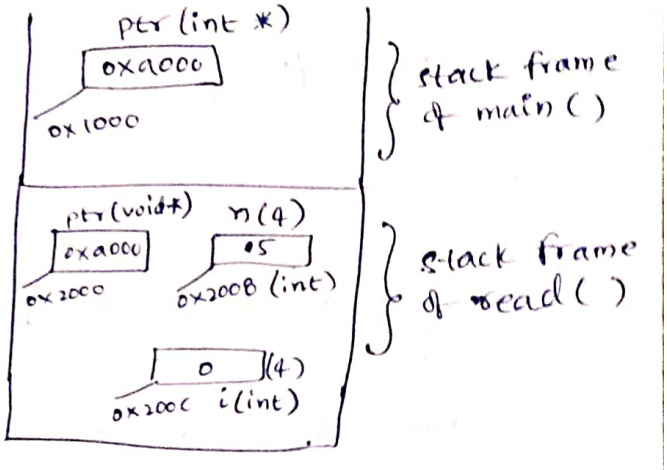


Requesting memory for 5 elements & accessing.

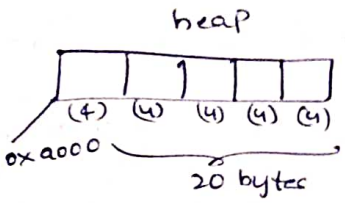
```
#define NM 5
main()
{
  int n, *ptr;
  ptr = (int *)malloc(NM * sizeof(int));   // 0xa000
  if (ptr==NULL)
  {
    PF("failed to allocate memory in heap\n");
    exit(1);
  }
  read(ptr, NM);
  display(ptr, NM);
  free(ptr);
}
```

return type

- **void** ⇒ That func^c will not return any value
- **void *** ⇒ That func^c will return the address which needs to be type casted. We don't know the type of data present at that address.




```
⇒ void read(void *ptr, int n)        // int *ptr
  {
    int i=0;
    for(i=0; i<n; i++)
    {
      PF("Enter the i/p\n");
      SF("%d", &((int *)ptr)[i]);    // &ptr[i]
    }
  }
```




```
⇒ void display(void *ptr, int n)
  {
    int i=0;
    for(i=0; i<n; i++)
    PF("%p - %d\n", &((int*)ptr)[i], ((int*)ptr)[i]);   // *((int*)ptr)+i)
  }
```

[ ] → *

⇒ main()
{
unsigned int x = 0xaabbccdd;
int i;
void * ptr = &x;

// byte by byte

for (i=0; i<4; i++)

PF("%p - %x\n", &((unsigned char *)ptr)[i], ((unsigned char *)ptr)[i]);

// 2 bytes

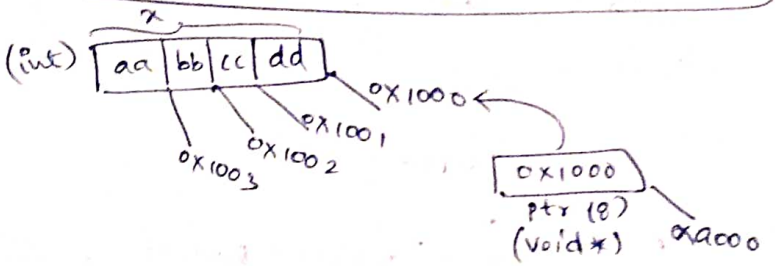
for (i=0; i<2; i++)

printf("%p - %x\n", &((unsigned short int *) ptr)[i], ((unsigned short int *) ptr)[i]);

}

(*) By using void pointer

accessing one byte & 2 bytes from 4 bytes



(*) Without using pointers

unsigned int x = 0xaabbccdd

int i;

for (i=0; i<4; i++)

// byte by byte printf("%p - %x\n", &((unsigned char *) &x)[i], ((unsigned char *) &x)[i]);

for (i=0; i<2; i++)

// 2 bytes PF("%p - %x\n", &((unsigned short int *) &x)[i], ((unsigned short int *) &x)[i]);

Assignment

→ Yesterday's prgm using void pointers

→ unsigned int x = 0x11223344;
" " y = 0xaabbccdd;
" " z;

o/p = z = 0xaa22cc44;

Using only pointers.

①

```
#include <stdio.h>
#include <stdlib.h>
void read_inputs (void *ptr, int n);
void display_inputs (void *ptr, int n);
void highest_digit (void *ptr, int n);
void second_highest (void *ptr, int n);
void paint_binary (void *ptr, int n);
void ascending_order (void *ptr, int n);
int main( )
  {
    unsigned int n, * ptr, opt;
    PF(" enter n value (no. of elements) \n");
    SF ("%d", &n);
    Ptr = (int *) calloc (n, size of (int));
    if (ptr==NULL)
    {
      PF (" failed to allocate memory in heap \n");
      exit (1);
    }
  while (1)
  {
  __fpurge(stdin);
  PF(" MENU \n 0. exit \n 1. read inputs \n 2. display_inputs \n
  3. highest_digit \n 4. second_highest \n 5. paint binary \n
  6. ascending_order \n");
  SF ("%d", &opt);
  switch (opt)
    {
      case 0 : exit(0);
      case 1 : PF("enter the n values \n");
               read_inputs (ptr, n);
               break;
      case 2 : PF ("displaying the inputs \n");
               display_inputs ((int *)ptr), n);
               break;
```

```
        case 3: highest_digit ((int *)ptr), n);
                break;
        case 4: second_highest ((int *)ptr), n);
                break;
        case 5: print_binary(((int *)ptr), n);
                break;
        case 6: ascending_order((int *)ptr), n);
                break;
        default : pf (" Invalid option\n");
        }
    }
    free(ptr);
}
```

① ⇒
```
void read_inputs (void *ptr, int n)
{
    int i;
    printf("enter the i/p's \n");
    for (i=0; i<n; i++)
        scanf ("%d", &((int *)ptr)[i]);
}
```

② ⇒
```
void display_inputs ( void *ptr, int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%p - %d\n", &((int *)ptr)[i], ((int*)ptr)[i]);
}
```

③ ⇒
```
void highest_digit (void *ptr, int n)
{
    int i, max=0;
    for(i=0; i<n; i++)
    {
        if (max < ((int*)ptr)[i])
            max = ((int*)ptr)[i];
            PF("highest digit is %d\n", max);
}
```

④ ⇒
```
void second_highest (void *ptr, int n)
{
    int big=0, i=0, _2big=0;
    for (i=0; i<n; i++)
    {
        if(((int*)ptr)[i] > big)
        {
            _2big = big;
            big = ((int*)ptr)[i];
        }
        PF("second highest is %d\n", _2big);
}
```

Else if (((int*)ptr)[i] > _2big)
_2big = ((int*)ptr)[i];

⑤
```
void print_binary (void *ptr, int n)
{
    int i, j=1, rem=0, sum=0, temp;
    for (i=0; i<n; i++)
    {
        temp = ((int*)ptr)[i];
        sum = 0;
        rem = 0;
        j = 1;
        while (temp>0)
        {
            rem = temp%2;
            temp = temp/2;
            sum = sum + rem*j;
            j = j*10;
        }
        PF("%d - %d\n", ((int*)ptr)[i], sum);
}
}
```

⑥ ⇒ void ascending_order (void *ptr, int n)

```
{
    int i, j; temp
    for (i=0; i<n ; i++)
    {
        for (j= i+1 ; j<n ; j++)
        {
            if (((int *)ptr)[i] > ((int *)ptr)[j])
            {
                temp = ((int *)ptr)[i];
                ((int *)ptr)[i] = ((int *)ptr)[j];
                ((int *)ptr)[j] = temp ;
            }
        }
        PF("%d\n", ((int *)ptr)[i]);
    }
}
```

②

```
#include<stdio.h>
int main()
{
    unsigned int x = 0x11223344;
    unsigned int y = 0xaabbccdd;
    unsigned int z, i;
    char *ptr1 = &x;
    char *ptr2 = &y;
    char *ptr3 = &z;
/*  ptr3[0] = ptr1[0];
    ptr3[1] = ptr2[1];
    ptr3[2] = ptr1[2];
    ptr3[3] = ptr2[3];
    printf("z = %x\n", *((int *)ptr3)); */
```

(I)

```
/*  for (i=0; i<4; i++)
    {
      if (i%2==0)
          ptr3[i] = ptr1[i];
      Else
          ptr3[i] = ptr2[i];
    }
    PF("z = %d\n", *((int *)ptr3));   */

    for (i=1; i<4; i=i+2)
    {
      ptr3[i] = ptr2[i];
      ptr3[i-1] = ptr1[i-1];
    }
    Printf("z = %d\n", *((int *)ptr3));
}
```

II

III

01/11/2022

* Arrays :-

⇒ Arrays are used to store same type of data elements in continuous memory locations.

⇒ Arrays and pointers comes under user defined

⇒ Arrays, pointers, structures & Unions comes under Non-primitive (user defined) data types.

⇒ char, int, float, double comes under primitive data types (pre-defined)

⇒ Arrays are constructed or designed by the concept of arithmetic operations on pointers.

* Syntax of array declaration :-

< data type > < array name > [n] ;

n → no. of elements

↓

It specifies:
→ type of data
→ each element size
→ range of values based on sign qualifiers.

⇒ This statement will allocate the memory statically. Compiler will decide the size of this array based on No. of elements & data type.

$$\text{Size} = n * \text{size of (data type)}$$

* Integer arrays :-

⇒ Uninitialized local array contains garbage value.

⇒ Memory is allocated in stack frame of corresponding fun^c.

⇒ Scope ⇒ Fun^c scope (or) Block scope.

⇒ Uninitialized global array memory gets allocated in bss segment.

⇒ Initialized global array memory gets allocated in data segment.

⇒ Global array life is till the end of the process.

⇒ Uninitialized global array contains zero

⇒ Global arrays will have file scope.

⇒ If we apply constant keyword for Global arrays it goes to rodata segment.

⇒ Array name will always return the starting base address of the array.

⇒ Array elements can be accessed by using array name along with index values. Array index always starts with zero (0).

⇒ At run time or execution time we cant modify the array base address.

⇒ Array's index starts with zero (0), cuz it follows Arithmetic operations on pointers.

arr[0] ⇒ *(arr + 0 * sizeof(int))

(or)

*arr = *(0xa000 + 0) = *(0xa000)

We can access the data by 5 ways using pointers

To access address using pointers

① *ptr, ptr++ ⇒ Not recommendable

② ptr[i] — ② &ptr[i]

③ i[ptr] — ③ &i[ptr]

④ *(ptr+i) — ④ (ptr+i)

⑤ *(i+ptr) — ⑤ (i+ptr)

⇒ we can access the array elements in 4 ways.

Data	Address
① arr[i]	① &arr[i]
② i[arr]	② &i[arr]
③ *(arr+i)	③ (arr+i)
④ *(i+arr)	④ (i+arr)

*arr = valid

arr++ = Not valid.

⇒ size of (arr) ⇒ 20 bytes [It access complete 5 elements bytes]

⇒ size of (ptr) ⇒ 8 bytes [depends on architecture machine]

⇒ size of (&arr[i]) ⇒ 8 bytes.

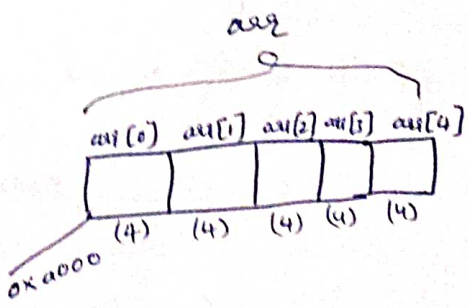
⇒ Array no. of Elements can be defined by using MACRO.

integer arrays

```
# define NM 5
main()
{
    int arr[NM];
```

Printf("%d\n", sizeof(arr)); (20)

Printf("%p\n", arr); 0xa000



```
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

(or)

```
scanf("%d", &arr[0]);
scanf("%d", &arr[1]);
scanf("%d", &arr[2]);
scanf("%d", &arr[3]);
scanf("%d", &arr[4]);
```

(or)

```
for(i=0; i<NM; i++)
{
  PF("enter the i/p\n");
  SF("%d", &arr[i]);
}
```

(or)

```
read (arr, NM);
display (arr, NM);
}
```

⇒

```
void read (int *ptr, int n)
{
    int i;
    PF("%d", sizeof(ptr));
    for(i=0; i<n; i++)
    {
        PF("enter the input\n");
        SF("%d", &ptr[i]);
    }
}
```

⇒

```
void display (int *ptr, int n)
{
    int i;
    for(i=0; i<n; i++)
    PF("%p - %d\n", &ptr[i], ptr[i]);
}
```

→ Reverse the elements in an array of 5 elements

① without using additional array
② By using additional array.

I/p = 10, 20, 30, 40, 50
O/p = 50, 40, 30, 20, 10.

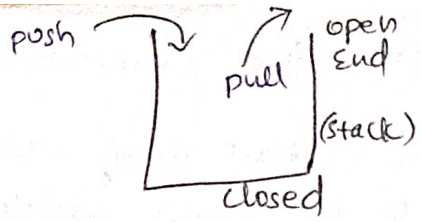
```
→ #include <stdio.h>
  int main()
  {
    int n, arr[n];
    PF("enter n value\n");
    SF ("%d", &n);
    read (arr, n);
    reverse_integer (arr, n);
  }

→ void read (int *ptr, int n)
  {
    int i;
    for (i=0; i<n; i++)
    SF("%d", &ptr[i]);
  }

→ void reverse_integer (int *ptr, int n)
  {
    int i, j, temp;
    for (i=0, j=n-1; i<j; i++, j--)
    {
      temp = ptr[i];
      ptr[i] = ptr[j];
      ptr[j] = temp;
    }
  }
```

* Assignment :-

① Find the second least element in a given array

② Implement stack by using arrays.

⊛ Stack :-



⇒ stack means last in first out.

⇒ Stack will have one end open and one end as closed

⇒ Adding the elements & deleting the elements can be done ~~from~~ only from open end.

⇒ Adding the elements to the stack is known as push operation.

⇒ Deleting the elements to the stack is known as pop op

⇒ The top of the element is always pointed by stack pointer or by reference point.

⇒ when we perform pop operation the top most element will be deleted.

⇒ ~~when~~ For every pop operation the reference point is decremented.

⇒ For every push operation the reference point is incremented.

⇒ For every push & pop operation, the reference point is updated by incrementing or decrementing.

(2) Implement stack by using arrays

```
int i=-1;
void push (int *ptr, int num);
void pop ( );
void display (int * ptr);
int search (int *ptr, int num);
main( )
 {
    int stack[5];
    int opt, num, pos;
while (1)
   {
    printf ("MENU \n 0. exit \n 1. push \n 2. pop \n 3. display \n 4. search \n");
    scanf ("%d", &opt);
    switch (opt)
     {
        case 0 : exit (0);
        case 1 : printf ("enter the value \n");
                 scanf ("%d", &num);
                 Push (stack, num);
                  break;
        case 2 : pop ( );
                 break;
        case 3 : display (stack);
                 break;
        case 4 : print ("enter the i/p \n");
                 scanf ("%d", & num);
                pos = search (stack, num);
                  if (pos == -1)
                   {
                     printf ("Element not found \n");
                     break;
                   }
                   printf("Element found at %d\n", pos);
                   break;
```

```
        default : printf (" Invalid option \n");
        }
    }
}
```

```
→ void push (int *ptr, int num)
  {
    if (i==4)
    {
      PF(" stack is full \n");
      return;
    }
    i++;
    ptr[i]= num;
  }
```

→ if (ptr==NULL)
  PF(" received invalid i/p \n");

```
⇒ void pop ()
  {
    if (i==-1)
    {
      printf (" stack is empty \n");
      return;
    }
    i--;
  }
```

```
⇒ void display (int *ptr)
  {
    int j;
    if (i==-1)
    {
      PF(" stack is empty \n");
      return;
    }
  for (j=0 ; j<=i ; j++)
    {
      PF("%d\n", ptr[j]);
    }
  }
```

```
⇒ int search (int *ptr, int num)
  {
    int j;
    if (i==-1)
    {
      printf (" stack is empty \n");
      return -1;
    }
  for (j=0; j<=i; j++)
    {
      if (ptr[j]==num)
        return j+1;
    }
    return -1;
  }
```

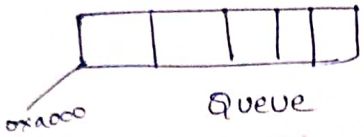
0

* Queue :-

⇒ It means FIFO (First in First out).

⇒ Adding the elements to the queue is ntg but insertion or insert operation

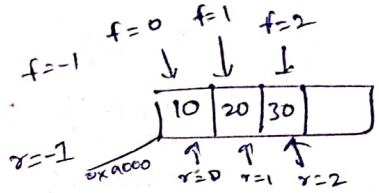
⇒ Deleting the elements to the queue is ntg but remove or delete operation



cases

-1 ← f index → delete → we cant Queue empty

-1 ← r index → insert → Queue full



⇒ **Insert**

1st operation

insert → rear index, front index ⇒ increment

2nd operation onwards

only rear index increment

⇒ **Delete**

Front index increment

front == rear ⇒ reset to -1.

03/11/22

```
#define SIZE 5
```

globally declared

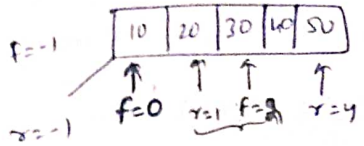
```
int f=-1;
int r=-1;
```

Queue → 5 Elements

→ insert
→ Delete
→ display.

⇒ void insert (int *ptr, int num)

```
{
    if (f==-1)
        f++ // r++;
    else if ((r==size-1) && (f==0)) || (r==f-1))
    {
        PF("Queue is full \n");
        return;
    }
    else if (r==size-1) // f>0
    {
        r=0;
        Ptr[r]=num;
        return;
    }
    else
    { r++; }
        Ptr[r]=num;
}
```



Cases

① Queue is Empty

② Queue is not empty

③ 3cases

i) F=0, r=N-1 ⟹ full

ii) if r=N-1 f>0
reset r=0;
update.

iii) if r=f-1
full

⇒ void delete ( )

```
{
    if (f==-1)
    {
        PF("Queue is empty \n");
        return;
    }
    else if (f==r)
    {
        f=r=-1;
        return;
    }
    else if (f==size-1)
    {
        f=0;
        return;
    }
    f++;
}
```

Cases

① Queue is Empty
if (f==-1)

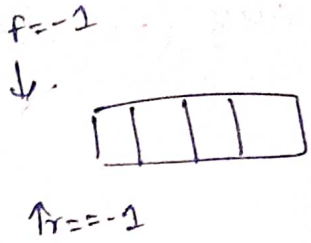
② Queue contains single element
if (f==r)
f==r=-1

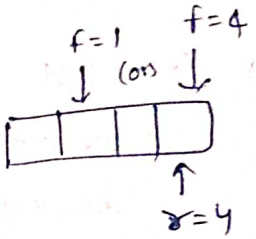
③ if (f=size-1) f is pointing last element (r<f)
f=0

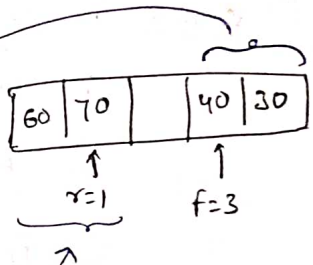
④ increment ++ to delete elements from Queue.

⇒ void display (int *ptr)

```
{
    int i;
if (f==-1)  // queue is empty
 {
   PF("queue is empty \n");
    return;
 }
if (f<=r)                  f → starting element
 {                         r → last element
   for (i=f ; i<=r ; i++)
      printf ("%d\n", ptr[i]);
 }
Else if (r<f)
   {
    for (i=f ; i<= SIZE -1 ; i++)
     printf ("%d\n", ptr[i]);
    for (i=0 ; i<=r ; i++)
     printf ("%d\n", ptr[i]);
   }
}
```



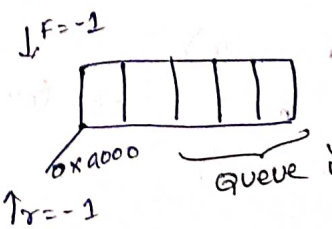


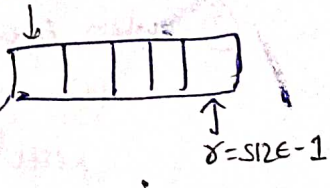


Explanation (Flow of Execution)

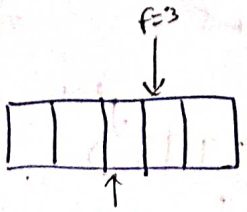
① Insert

```
void insert (int *ptr, int num)
 {
   if (f==-1)
     f++;
Else if ((r==SIZE-1)&&(f==0)) || (r==f-1)
```



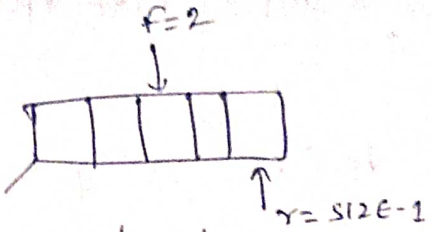


⇒ Queue is FULL



if r is less than 1 index w.r.t F
queue is Full.

else if (r == SIZE-1)



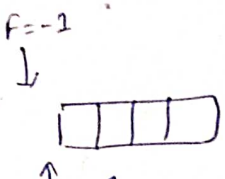
→ when f > 0
& need to insert some elements we need to make/equate/reset rare index to 0

⇒ To insert elements normally,
- we need to increment both front & rare index during 1st operation
- From next operation we have to increment only rare index to insert elements in queue.

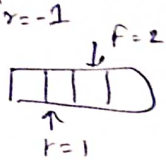
✱ Insert

cases :- case-1

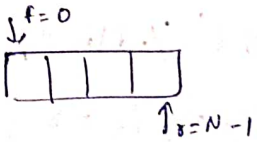
① when f=-1, r=-1 (queue is empty)



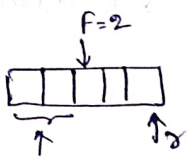
② when f & r doesn't lies at -1



③ i) f=0, r=N-1
queue is full

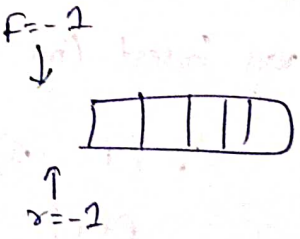


ii) r = N-1 (when f>0)
Set r=0

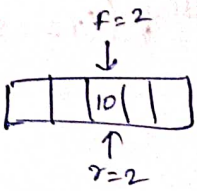


✱ Delete

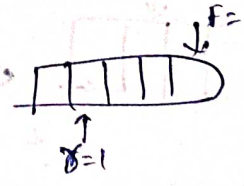
① if (f == -1)
or
(r == -1)
⇒ queue is empty



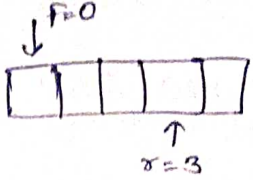
② when queue contains single element
(f == r)
Reset it to -1 to delete that single element



③ If f = SIZE-1
reset it to zero (f=0)



④ To delete elements from Queue



(f++)

↓

To delete element from Queue.

⇒ Ascending order :-

```
int ascending (int *ptr, int n)
{
    int i, j, temp;
    if (f==-1)
    {
        PF("Queue is empty\n");
        return -1;
    }
    else if (f<=r)
    {
        for (i=f; i<=r; i++)
        {
            for (j=i+1; j<=r; j++)
            {
                if (ptr[i] > ptr[j])
                {
                    temp = ptr[j];
                    ptr[j] = ptr[i];
                    ptr[i] = temp;
                }
            }
            printf ("%d\n", ptr[i]);
        }
    }
    else if (r<f)
    {
        for (i=0; i<n; i++)
        {
            if ((i<r) || (i>=f))
            {
                for (j=i+1; j<n; j++)
                {
                    if ((i==r) || (j>=f))
                    {
                        if (ptr[i] > ptr[j])
                        {
                            temp = ptr[i];
                            ptr[i] = ptr[j];
                            ptr[j] = temp;
                        }
                    }
                }
            }
            PF("%d\n", ptr[i]);
        }
    }
```

⊛ Character arrays :-

→ Character arrays are used to store string constants

String constants:
- Group of characters enclosed with in pair of double quoutes (" ")
- Group of characters enclosed individually within pair of curly braces {'a', 'b', 'c', '\0'}.

"abc" (or) {'a', 'b', 'c', '\0'}.

↓ If we write like this, we don't need to represent null character (\0)

If we write like this, we need to include null character (\0) also.

⇒ char str[20] = "abc";

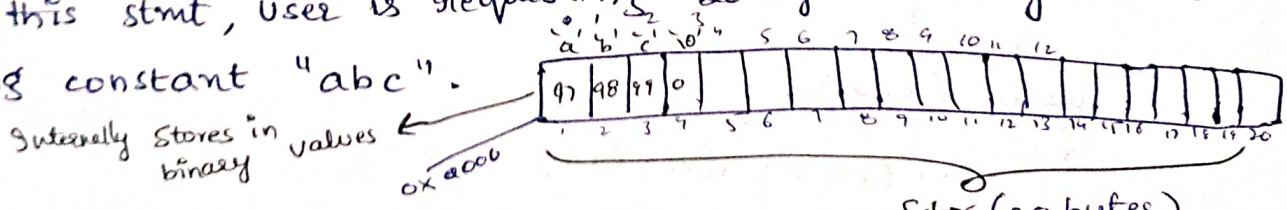
size of array = n * size of (datatype)
= 20 * 1
= 20

Printf ("%d", sizeof (str)); ⇒ 20 bytes

Printf ("%d", sizeof ("abc")); ⇒ 4

⇒ char str[20] = "abc";

⇒ In this stmt, user is requesting 20 bytes memory to store string constant "abc".



→ The name given to this 20 bytes is Str.
So, when we use str i.e., array name, it will return the 20 bytes starting base address.

⇒ In each byte it will store corresponding ASCII character's ASCII value in binary format.

⇒ This character array can be accessed by character by character or as a string. (by using base add) // By using index values

⇒ we can access character by character by using array name with index values.

⇒ To access the entire string we have to use array name

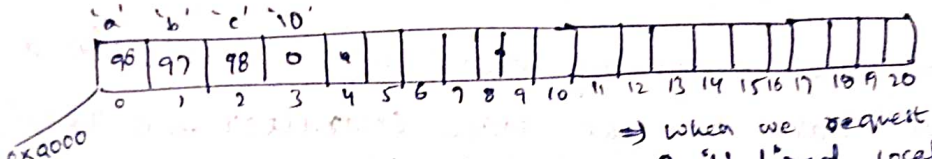
⇒ The strings which are directly passed to the function and the strings which are initialized by character pointer their strings memory is allocated in .rodata segment.

(*) How do you request the memory dynamically for a string constant.

```
Char *ptr;

Ptr = (char *) malloc (6 * size of (char));
```

①

```
char Str[20] = "abc";
```



⇒ address → entire string → by array name.

⇒ individual elements/characters → Array name [index].

Data ⇒ Read (access) / write (modify)

⇒ when we request to allocate memory statically ⇒ Initialized local variables/arrays goes to _____

②

```
Char *ptr = "Kumar";
```

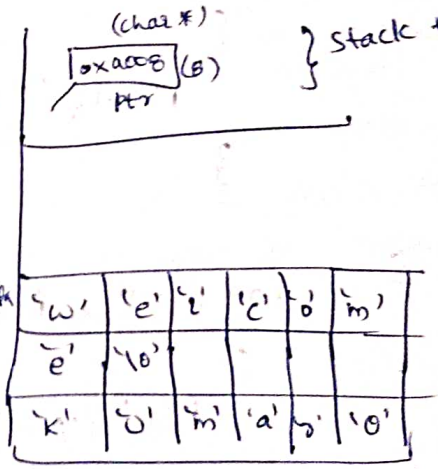
0x8008

```
printf ("welcome");
```

0x9000

⇓

Indirectly we are passing string's starting base address.



⇒ These strings go to .rodata segment

⇒ address → entire string → pointer

⇒ individual element/character → pointer [index]

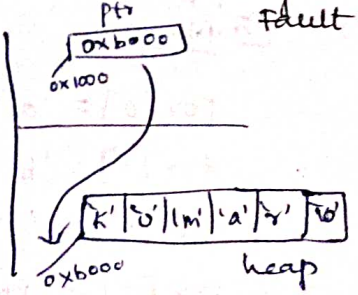
Data – Read (access) only

If we try to write (modify) ⇒ Segmentation fault error

③

```
ptr = (char *) malloc (6 * size of (char));
```

⇒ These strings, memory gets allocated in heap segment.



⇒ address → entire string → pointer.

⇒ individual element/character → pointer [index]

⇒ Data = Read (access) / write (modify).

⇒ To print the string constant by using printf function we have to use %s format specifier. This format specifier requires strings starting base address.

1) Printf ("%s", str); ⇒ abc.
0xa000

- It prints character by character until it finds NULL (\0) characters with reference to starting address.

2) printf ("%s", ptr); ⇒ welcome.
0xa000

3) Printf ("%s", ptr);

⇒ when we are accessing individual elements in a character array, we have to use NULL character for teminating the loop.

①
```
for (i=0 ; str[i] != '\0' ; i++)
  {
    printf("%p-%c\n", &str[i] , str[i]);
  }
```

②
```
for (i=0 ; ptr[i] != '\0' ; i++)
  {
    PF("%p-%c-%d\n", &ptr[i], ptr[i], ptr[i]);
```

③
⇒ we can update character strings in ③ ways

① Manually
```
ptr[0] = 'a' ;
ptr[1] = 'b' ;
ptr[2] = 'c' ;
ptr[3] = '\0' ;
```

(or)

② ← Keyboard through Terminal
```
scanf("%s", ptr);
```
abc↵

③ strcpy (ptr, "abc");

① ⇒ str[0] = 'k';
str[1] = 'u';
str[2] = 'm';
str[3] = 'a';
str[4] = 'r';
str[5] = '\0';

(or)

scanf("%s", str);

or

strcpy(str, "kumar");

② ⇒ In directly passed strings we cant update character strings.

⇒ int arr[5]; ⇒ uninitialized array / declaration

⇒ int arr[5] = {10, 20, 30, 40, 50} ⇒ initialized array / defination.

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

⇒ This is valid statement. Based on initialed values the compiler will decide the size of array.

⇒ int arr[5] = {1, 2}; ⇒ 20 bytes.

- In this case, first 2 array elements are updated with 1 & 2. remaining elements are filled with zeroe's.

⇒ int arr[ ] = {1, 2}; ⇒ 8 bytes. (compiler will decides the size of array based on element number & data type).

⇒ int arr[ ]; ⇒ Invalid (compilation error, cuz compiler will not be able to decide the size of array even the data type is mentioned).

⇒ int arr[2] = {3, 4, 5, 6, 7};

arr[2] ↓ 8 bytes

{3, 4, 5, 6, 7} ↓ First 2 values are updated in 8 bytes memory.

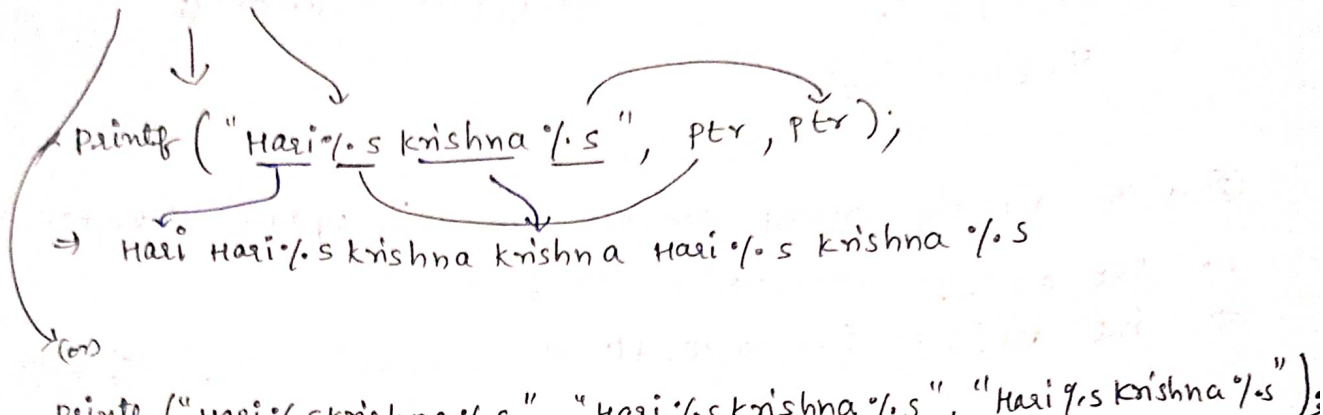
⇒ char str[ ] = "kumar"; 6 bytes

⇒ char str[10] = "kumar"; 10 bytes

⇒ char str[10] = {'k', 'u', 'm', 'a', 'r'};

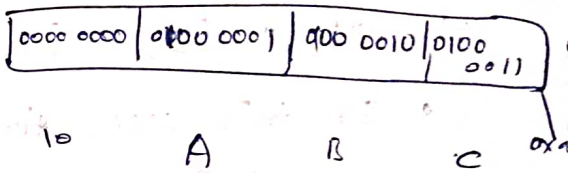
⇒ char str[10] = {'\0'}; ⇒ it prints '\0' 10 tymes.

⇒ main()
{
char *ptr = " Hari %.s krishna %.s" ;
printf (ptr, ptr, ptr);
}



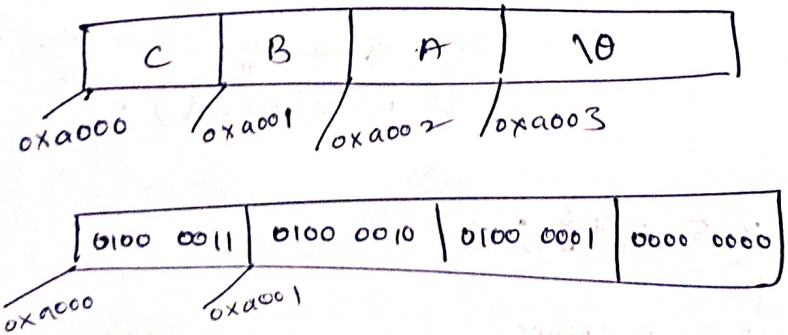
⇒ unsigned int x = 0X41424300 ;
printf (&x); ⇒ o/p
ABC

① unsigned int x = 0X 00414243;
printf(&x);



o/p = CBA

② char *ptr = " CBA " ;



printf (ptr);
↑
0xa000

04/11/2022

* String Manipulation Functions :-

⇒ The operations that we are performing on strings they are nothing but string manipulation functions.

⇒ String copy function :- # include <string.h>

```
char * strcpy(char *dst, const char *src);
```
⇒ fun^c definition

⇒ This function will copies the string from source address to destination address including null character. After copying the data, it will return the destination memory base address.

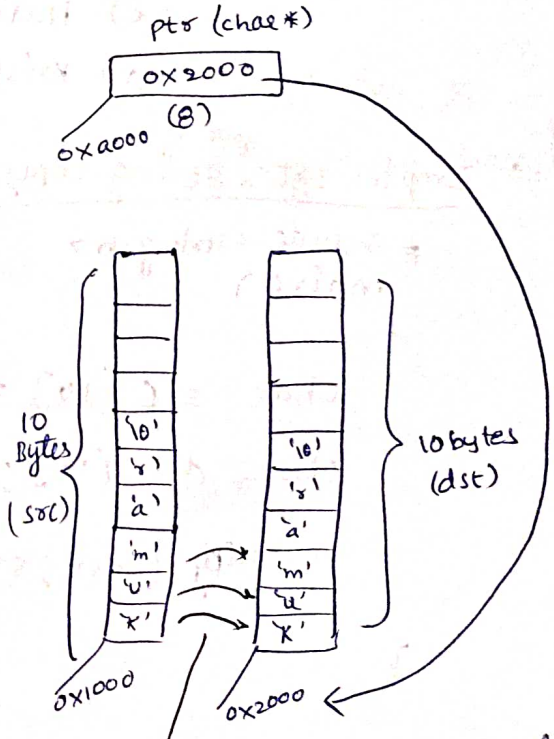
⇒ This fun^c will not verify the source or ~~dest~~ destination memory sizes.

⇒ The developer has to take care of source & destination memories

⇒ Destination memory size should be greater than or equal to source memory size.

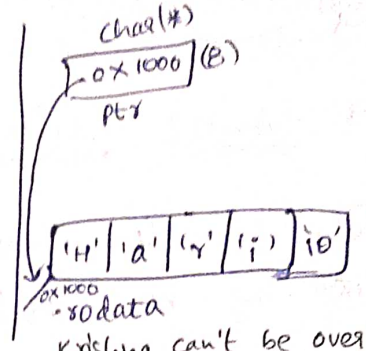
```
#include <string.h>
main()
{
  char *ptr;
  char dst[10];
  Char src[10] = "kumar";
ptr = Strcpy(dst, src);   (0x2000)
  Printf("%p-%s", dst, dst);
        (or)
  Printf("%p-%s", ptr, ptr);
}
```

⇒ This will return destination array starting base address.



⇒ It copies byte by byte from source memory to destination memory

⇒
```
main()
{
    char *ptr = "Hari";
    strcpy(ptr, "Krishna");
    printf("%s", ptr);
}
```

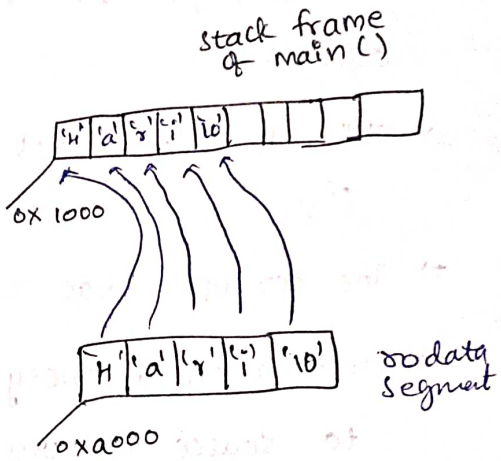


Krishna can't be overwrite in rodata segment cuz it is readonly data segment we can't modify the content of rodata segment so kernal will send a signal SIGSEGV to kill that process or application

a) compilation error
b) segmentation error
c) Hari
d) Krishna

⇒
```
main()
{
    char dst[10];
    strcpy(dst, "Hari");
    printf("%s\n", dst);
}
```

dst → memory allocated in stack frame of main()
"Hari" → goes to rodata segment
dst = 0x1000, "Hari" = 0xa000



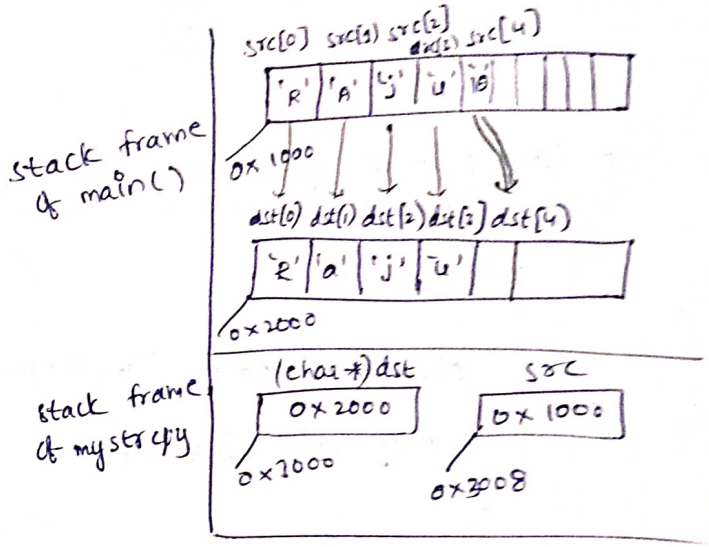
O/P

a) compilation error
b) segmentation error
c) Hari
d) Krishna

⇒ Implement your own strcpy function :-

```
#include <string.h>
main()
{
    char src[10] = "Raju";
    char dst[10];
    mystrcpy(dst, src);
}
```

```
char * mystrcpy ( char *dst, char *src)
{
    int i;
    for (i=0 ; src[i] != '\0' ; i++)
        dst[i] = src[i];
    dst[i] = '\0' ; // dst[i] = src[i];
    return dst;
}
```



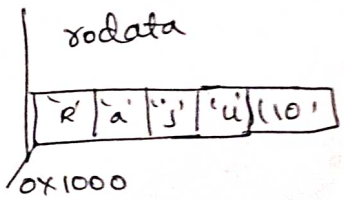
* String length function :-

```
size_t   strlen (const char * src);
```
↑ return type

⇒ This func will take string's base address as i/p & returns the length of the string excluding null character

```
main()
{
    int len;
    len = strlen("Raju");
    printf("%d", len);
}
```

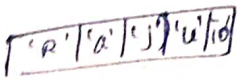
(4) → "Raju" (0x1000)



o/p

4 (excluding null character)

⇒ main()
{
char src[10] = "Raju";
int len;
len = strlen(src);
Printf("%d", len);
}



stack frame of main fun^c.

o/p
4

⇒ Implement your own string length function :-

```
main()
{
  char src[10] = "welcome";
  mystrlen(src);
}

size_t mystrlen(char *src)
{
  int i;
  if (src == NULL)
    return 0;
  for (i=0; src[i] != '\0'; i++);
    return i;
}
```

↳ It will make loop till the condition becomes false

* String compare :-

⇒ Comparision of two strings can be done by string compare function. so, this fun^c will compare the two strings from the given addresses and returns zero (0) if they are same and a non-zero value if they are not same.

```
size_t  strcmp (const char * s1, const char * s2);
```

⇒
```
main ()
{
  int i;
  i = strcmp ("Raju", "Raju");
    if (i==0)
      printf ("both are same\n");
    else
      printf ("Not same");
}
```

⇒ Implement your own string compare function :-

```
int mystrcmp (char * s1, char * s2)
{
  int i, j, ind;
  if ((s1==NULL) || (s2==NULL)
    return -1;
  i = strlen (s1);
  j = strlen (s2);
  if (i!=j)
    return (i-j);
for (ind=0; s1[ind]!='\0'; ind++)
  {
    if (s1[ind]!=s2[ind])
      return (s1[ind]-s2[ind]);
  }
return 0;
}
```

(or)

```
for (ind=0 ; (s1[ind] != '\0') || (s2[ind] != '\0') ; ind++)
{
    if (s1[ind] != s2[ind])
        return (s1[ind] - s2[ind])
}
return 0 ;
```

⇒ Implement string reverse function :-

```
char * str rev (char * src)
{
    int i, l ;
    char temp ;
    if (src == NULL)
      return NULL ;
    l = strlen (src); -1 ;
    for (i=0 ,l ; i<l ; i++ ,l--)
      {
          temp = src[i] ;
          src[i] = src[l] ;
          src[l] = temp ;
      }
    return src ;
}
```

* convert ASCII character to integer :-

"123" → 123

```
int ascii_int (char *src)
{
    int i, sum = 0 , temp;
    if (src == NULL)
        return -1;
```

```
for (i=0; src[i]!='\0'; i++)
    {
      if ((src[i]>='0') && (src[i]<='9'))
          {
            temp = src[i] - 48;
            sum = sum * 10 + temp;
          }
      else
       {
         printf ("Invalid input\n");
         return -1;
       }
    return sum;
}
```

Assignment ⇒ (2nd least number)

```
#include <stdio.h>
main()
{
  int i, n, min, smin, max=0;
  printf ("enter no. of elements\n");
  scanf ("%d", &n);
  int arr[n];
  for (i=0; i<n; i++)
    {
      printf ("enter the i/p\n");
      scanf ("%d", &arr[i]);
    }
  for (i=0; i<n; i++)
    {
      if (max<arr[i])
        max = arr[i];
    }
  min = max, smin = max;
  for (i=0; i<n; i++)
  {
    if (arr[i]<min)
      {
        smin = min;
        min = arr[i];
      }
    else if ((arr[i]<smin)&&(arr[i]!=min))
        smin = arr[i];
  }
  PF ("% second least is %d", smin);
}
```

I/p :-

10 20 10 10 50

o/p ⇒ second least is 20.

(*) **Assignment :-** By using fun' definitions

① int ⇄ ASCII characters conversion

② String or stmt ⇒ print each character repeated how many times.

③ Verify given string is palindrome or not.

④ search for the string in a given stmt.

⑤ string concatination

"Raju" "kumar"

O/P ⇒ "Raju kumar"

① integer to ASCII character

```
#include <stdio.h>
void int_ascii (int n);
int main()
{
    int n;
    printf("Enter the i/p\n");
    scanf("%d", &n);
    int_ascii(n);
}
```

⇒

```
void int_ascii (int n)
{
    char arr[100];
    int i, count=0, rem, temp, sum=0;
    temp=n;
    while (temp>0)
    {
        temp = temp/10;
        count++;
    }
    sum=n;
    for (i=0; sum!=0; i++)
    { rem=sum%10;
      sum= sum/10;
      arr[count-1-i]=rem+48;
    }
    arr[i]='\0';
    printf("%s\n", arr);
}
```

I/P

123

O/P = ASCII

123.

05/11/2022

③ Palindrome or not

```
main()
{
  char src[10] = "liril";
  palindrome(src);
}
```

⇒
```
void palindrome(char *ptr)
{
    int stat;
    char *dst;
  dst = (char *) malloc (mystrlen(ptr)+1) * sizeof(char));
                          i
    if (dst == NULL)
    {
      PF ("Failed to allocate memory in heap \n");
       exit(1);
    }
  mystrcpy (dst, ptr);
  mystrrev (dst);
  stat = mystrcmp (dst, ptr)
     if (stat == 0)
         printf ("palindrome");
     else
         printf ("Not a palindrome\n");
  free (dst);
}
```

⇒
```
int mystrlen (char *ptr)
{
   int i;
   for(i=0; ptr[i] != '\0'; i++);
      return i;
}
```

⇒
```
char * mystrcpy (char *dst, char *
                                src)
{ int i;
  for(i=0; src[i] != '\0'; i++)
     dst[i] = src[i];
     dst[i] = src[i];
  return dst;
}
```
or `'\0';`

To update null char after loop gets terminate

⇒ Void mystrrev(char *dst)

```
{
    char temp;
    int i,j;
    j = mystrlen(dst) - 1;
  for(i=0, j; i<j; i++, j--)
   {
      temp = dst[i];
      dst[i] = dst[j];
      dst[j] = temp;
   }
}
```

⇒ int mystrcpy(char *dst, char *src)

```
{
   int i;
for(i=0; (dst[i]!='\0')||(src[i]!='\0'); i++)
   {
     if(dst[i] != src[i])
       return 1;
   }
return 0;
}
```

(or)

⇒

```
#include <stdio.h>
#include <string.h>
char pal_or_not(char *ptr);
int main()
{
    char arr[100], ret;
    printf("enter string\n");
    scanf("%s", &arr);
    ret = pal_or_not(arr);
    if(ret==0)
      Printf("palindrome\n");
    else
      PF("Not a palindrome\n");
}
```

Ip - olr

MISS - Not a palindrome

MOM - palindrome.

```
char pal_or_not(char *ptr)
{
   int i,l;
   char temp, cpy[100];
   for(i=0; ptr[i]!='\0'; i++)
   {
     cpy[i] = ptr[i];
   }
   cpy[i] = '\0';
for(i=0; cpy[i]!='\0'; i++);
// Printf("length is %d\n", i);
   l = i-1;
for(i=0, l; i<l; i++, l--)
   {
     temp = ptr[i];
     Ptr[i] = ptr[l];
     ptr[l] = temp;
   }
for(i=0; ptr[i]!='\0'; i++)
   {
     if(cpy[i]!=ptr[i])
       return -1;
   }
return 0;
}
```

* Function pointer :-

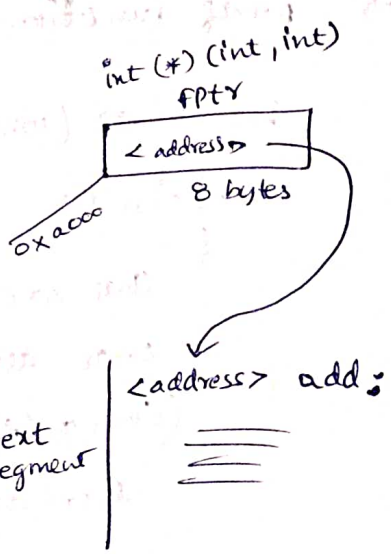
⇒ If we want to invoke a fun^c by using its address we have to use fun^c pointer.

Steps :-

① create, declare or define a fun^c pointer of same type of function int (*fptr) (int, int);

② Assign the functions base address to that fun^c pointer. We can get the function's base address by using fun^c name. fptr = add;

③ Invoke the fun^c by using fun^c pointer by passing valid inputs. res = add (a, b);




```
main()
{
   int res;
                    // create fun^c pointer.
   int (*fptr) (int, int);
    // Assign fun^c base address
    fptr = add;
    printf ("%p\n", fptr);
                or
    printf ("%p\n", add);

   // res = add (a,b)
// calling the fun^c w.r.t base address (fun^c pointer)
        res = fptr (a,b);
    Printf ("%d\n", res);
}
```

⇒ `char* strcpy(char* dst, const char *src);`

```
main()
{
  char src[10] = "hello";
  char dst[10];
```

create → `char* (*fptr)(char*, char*);`

assign → `fptr = strcpy;`

Invocation → `fptr(dst, src);`

```
// strcpy(dst, src);
  printf("%s\n", dst);
}
```

⇒ printf function

`int printf(const char*, ...);`

```
main()
{
  char src[10] = "hello";
  char dst[10];
  (char* (*)fptr)(char*, char*);
  fptr(dst, src); → (3)
  fprint("%s\n", dst);
}
```

⇒ `int (*fprint)(char*, ...) = printf;` ①&②

⇒ Passing a function's base address as an input to another func is known as call back func.

⇒ `void display(int*, int, int(*fptr)(char*, ...));`

```
main()
{
  int arr[5] = {10, 20, 30, 40, 50};
  display(arr, 5, printf);
}
```

```
void display (int *ptr, int n, int (*fptr)(char *, ...))
{
    int i;
    for(i=0; i<n; i++)
    {
    // Printf ("%d\n", ptr[i]);
       fptr ("%d\n", ptr[i]);
    }
}
```

⇒ when we write call back fun^c we have to consider the inputs even for that particular fun^c also. Those i/p's should be passed as the inputs to the call back function

```
⇒ void int_ascii (int num, char *dst, int (*fptr)(int)
                  (123)    (src)        (count)
{
    int j, rem;
    ③
    j = fptr (num);
    dst[j] = '\0';
    for (j=j-1; j>=0; j--)
    {
        rem = num % 10;
        dst[j] = rem + 48;
        num = num/10;
    }
}
```

'1'	'2'	'3'	'\0'
0	1	2	3

```
if (rem>=0)&&
(rem<=9)
```

(or)

```
while (num>0)
{
    j--;
    rem = num % 10;
    dst[j] = rem + 48;
    num = num/10;
}
```

```
int count (int num)
          (123)
{
    int c=0;
    while (num>0)
    {
        c++;
        num = num/10;
    }
    return ;
    ③
```

⇒ How do you invoke a function using void pointer

```
main()
{
  char src[10] = "hello";
  char dst[10];
  ~~char *fpt~~  void *ptr;
        ptr = strcpy;
  ((char *(*)(char *, char *))ptr)(dst, src);
  printf("%s\n", dst);
}
```

⇒ Using void pointer write display prgm

printf() ⇒ int printf(const char *, ...)

```
main()
{
  int arr[5] = {10,20,30,40,50};
  display(arr, 5, printf);
}
```

It passes printf starting base address.

⇒ `void display(int *ptr, int n, int (*fptr)(const char *, ...))`

fun^c pointer for printf()

```
{
  int i;
  for(i=0; i<n; i++)
    fptr("%d\n", ptr[i]);
}
```

⇒ `void display(int *ptr, int n, void *vptr)`

void pointer

```
{
  int i;
  for(i=0; i<n; i++)
(int (*)(const char *, ...))vptr)("%d\n", ptr[i]);
}
```

Type casting to behave ~~void~~ pointer as fun^c pointer

(2) // pgm to print each character repeated howmany times.

```
#include <stdio.h>
#include <string.h>
void char_repeated (char *ptr);
int main ()
{
    char arr[100];
    PF("Enter the string\n");
    sf (" %[^\n]s ", &arr);
    char_repeated (arr);
}
```

⇒

```
void char_repeated (char *ptr);
{
    int i, j, len, count = 1;
    len = strlen (ptr);
    for (i=0; i< len; i++)
    {
        if (ptr[i] == ' ')
            continue;
        count = 1;
        if (ptr[i] != 1) ═══ ?
        {
            for (j= i+1; j<=len; j++)
            {
                if (ptr[i] != ptr[j])
                {
                    count ++;
                    ptr [j]= 1; ═══ ?
                }
            }
            Printf ("%c - %d\n", ptr[i], count);
        }
    }
}
```

I/p = Hi Hello world

o/p

H-1	o-2
i-1	w-1
e-1	r-1
l-3	d-1

④ //search for a given string in the given stmt is exist & print how many times the

⇒
```
#include <stdio.h>
#include <string.h>
int mystrcmp (char *src, char* dst, int l2);
void main()
{
    int stat=0, count=0, l1, l2, i;
    char src[100];
    char dst[100];
    printf("enter a string\n");
    scanf("%[^\n]s", src);
    fpurge(stdin);
    printf("enter the string to be searched\n");
    scanf("%[^\n]s", dst);
    l1 = strlen(src);
    l2 = strlen(dst);
    for(i=0; i<=(l1-l2); )
    {
        stat = mystrcmp(&src[i], dst, l2);
        if (stat==0)
        { count++;
          i = i+l2;
        }
        i++;
    }
    PF("%d\n", count);
}
```

⇒
```
int mystrcmp(char *src, char *dst, int l2)
{
    int i;
    for(i=0; i<l2; i++)
    {
        if (src[i] != dst[i])
            return 1;
    }
    return 0;
}
```

I/p
① Hi Hi Hello hi
② Hi
o/p
3 (times hi is repeated)

⑤ // string concatenation.

```
#include <stdio.h>
#include <string.h>
void concatenation (char *ptr1, char *ptr2);
int main()
{
    char a[100], b[100];
    printf ("Enter a string\n");
    scanf ("%[^\n]s", &a);
    -- fpurge (stdin);
    printf ("Enter a second string\n");
    scanf ("%[^\n]s", &b);
    concatenation (a,b);
}
```

⇒

```
void concatenation (char *ptr1, char *ptr2)
{
    int l1, l2, l, i, j;
    char temp [l+i];  // char temp[100];
    l1 = strlen (ptr1);
    l2 = strlen (ptr2);
    l = l1+l2;
    for (i=0; i<l1; i++)
        temp[i] = ptr1[i];
    for (j=0, i; i<l; i++, i++)
    {
        temp[i] = ptr2[j];
    }
    temp[i] = '\0';
    printf ("%s\n", temp);
}
```

O/p :-

→ Enter a string
Hi this is.

→ Enter second string
kumar

⇒ Hi this is kumar.

(*) Objdump tool commands :-

(1) To get the file headers of an object file (executable file)

```
$ objdump -f <executable file name>
                (.o file)
```

(2) To print the object specific file header content

```
$objdump -p <File name>
```

(3) To print the section header content of the file.

```
$objdump -h <File name>
```

(4) To print the all header content of file.

```
$ objdump -x <File name>
```

(5) To print the assembler content of ~~all~~ the section ~~of~~ capable ~~the file~~ of execution

```
$objdump -d <File name>
```

(6) To print the assembler content of all the section of the file

```
$objdump -D <file name>
```

(7) To print the complete content of all the section of the file

```
$ objdump -s <file name>
```

(8) To print the content of all the section after disassmble content of the file (.text)

```
$objdump -S <file name>
```

(*) readelf commands :-

(1) To display readelf help command

```
$ readelf
```
or
```
$ readelf --help
```

(2) To check whether a file is ELF file or not.

```
$ file elf_file
```

(3) To generate elf file using gcc compiler

```
$ gcc filename.c -o elf_file
```

(4) To display file headers of elf file

```
$ readelf -h elf_file
```

(5) To display information about different section of process address space

```
$ readelf -S elf_file.
```

(6) To display symbol table

```
$ readelf -s elf_file
```

(7) To display core notes

```
$ readelf -n elf_files
```

(8) To display relocation section.

```
$ readelf -r elf_file
```

(9) To display the dynamic section

```
$ readelf -d elf_file
```

(10) To get the version of readelf

```
$ readelf -v
```

(11) readelf -a ⇒ all [equivalent to specifying --header -- pgm header -- sections -- symbols -- relocs --dynamic --notes --version -- info -- arch -- specific --unwind -- section -- groups & histogram].

(12) readelf -l ⇒ pgm header / -- segments.

⇒ Display info contained in files section group if it has any.

(13) readelf -g → -- section groups

⇒ Display info contained in files section group if it has any.

(14) readelf -u → unwind

⇒ Display contents of the files in unwind section if has any info.

Void pointer :- (syntax)

(1) declaration (creating a pointer)
(2) initialization (assigning fun^c base address to void pointer)
(3) fun^c invocation of void pointer.

① Void *vptr;
② vptr = address (fun^c name);
③ Void (*) (int, int) vptr (x, y);

- vptr → pointer name
- Return type of fun^c
- (*) → it represents type of pointer
- (int, int) → Data types of formal arguments
- (x, y) → Actual arguments which we are passing