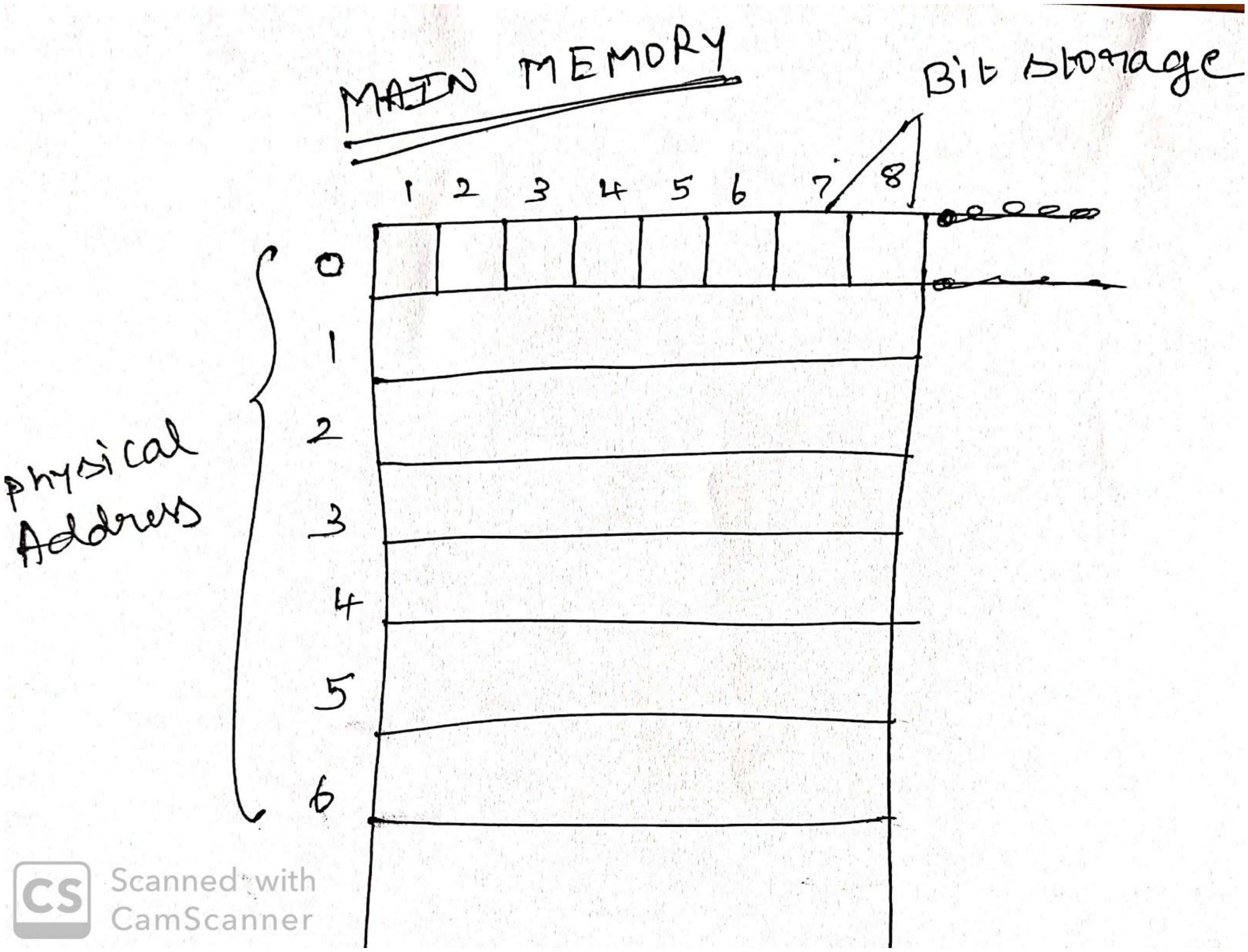
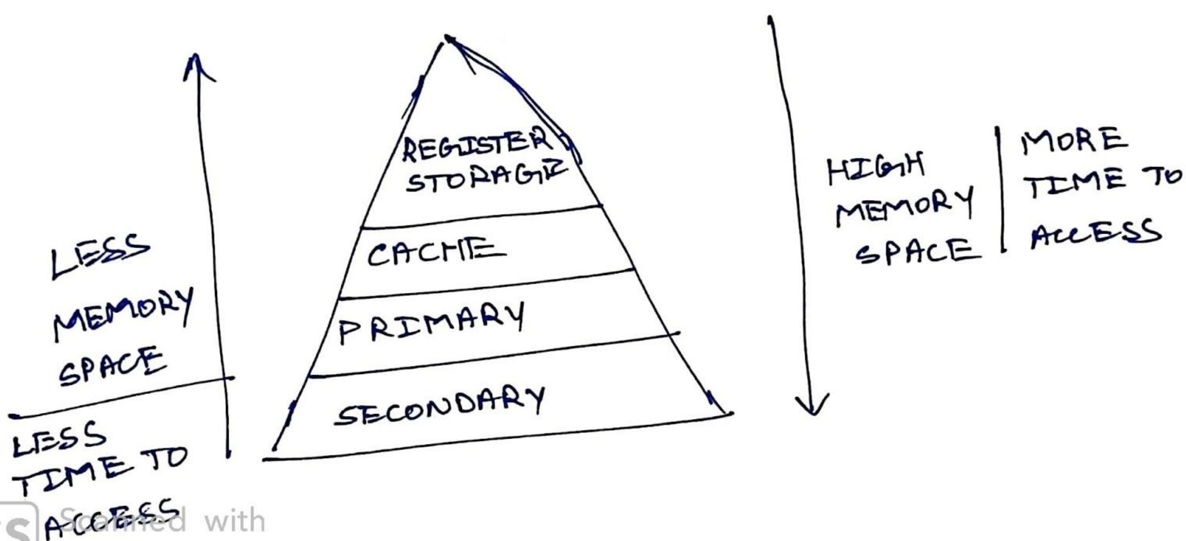
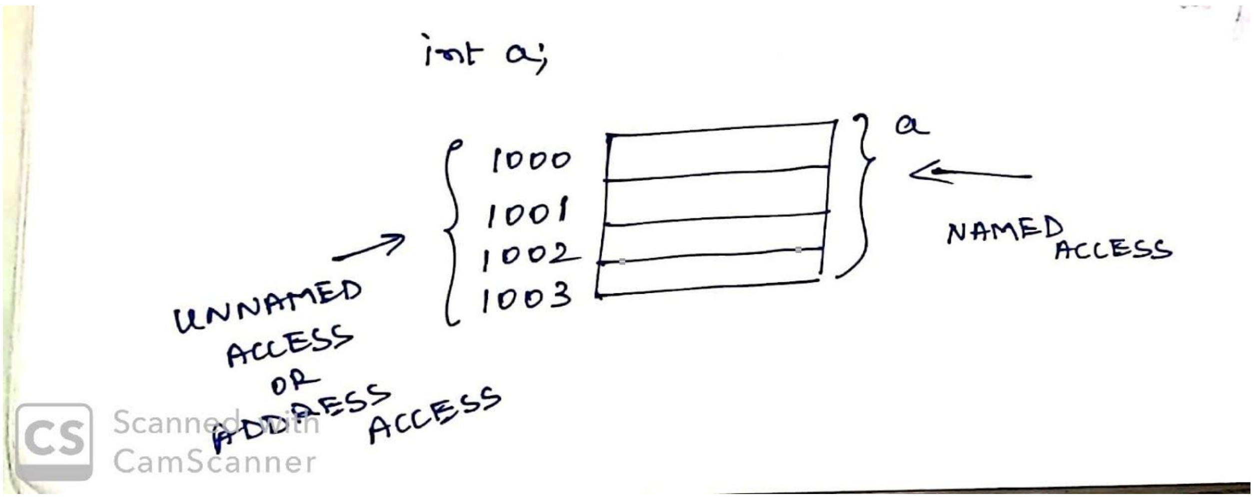
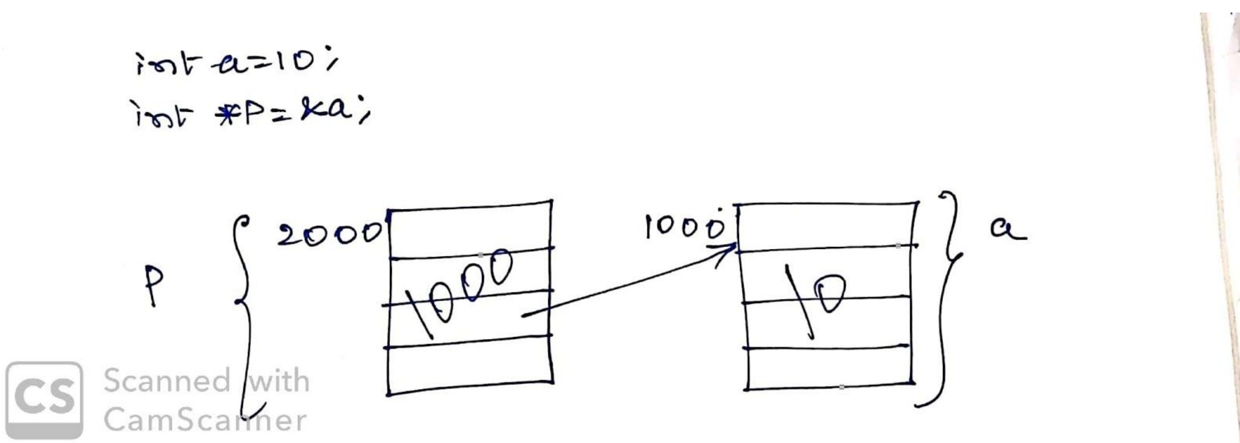

MAIN MEMORY
Bit storage
1 2 3 4 5 6 7 8
0
1
2
3
4
5
6
physical Address


REGISTER STORAGE
CACHE
PRIMARY
SECONDARY
LESS MEMORY SPACE
LESS TIME TO ACCESS
HIGH MEMORY SPACE
MORE TIME TO ACCESS





printf("%x", *p);

scanf("%d", p);

& - Address of operator

* - Dereference operator

Pointers
****
Pointer holds on to address / Pointer stores address
Variable stores information/data.
Address does not have a type, but the information stored in the addressed location has a type associated with it.

Irrespective of the type of pointer, all type of pointers will get 4 bytes of memory

```
GF:   <data-type>  *  <pointer-name>;
        int  * p;
```
p is considered as a pointer, which will hold on to integer variable's address.

```
        int  a;
        int *p;
        p = &a;
        printf("Content of pointer is %x %x",    *&p, p );

        scanf("%d",p);
        printf("%d",a);
        printf("Address of pointer p is %x", &p);
```

WAP to add two numbers using pointers
AIM : variables will be used to hold on to values, and pointers will be used to access variables value.

```
int main( )  {
        int a, b, sum;
        int *p=0, *q=0, *r=0;
// pointers having unwanted or garbage address is termed as DANGLING POINTERS
//it is a GOOD HABIT to initialize pointers to zero address, termed as NULL POINTERS
      printf("%x %x %x", p, q, r);
        p = &a;     q=&b;      r=&sum;
// &a is termed as "VALID ADDRESS" because in that address info can be stored.
        scanf("%d%d" ,  p, q);     // similar to scanf("%d%d", &a, &b);
//accepting values for a & b variables via p and q
        *r = *p + *q;
// *r means sum,   *p means a   and *q means b
        printf("%d + %d = %d", *p, *q, *r);
 }
```

Scope and lifetime
Scope is the region or area of a program where the identifiers are accessible or visible.
(Identifiers are variable name, constant name, function name etc.,)

Lifetime is the time period where the variables will be alive.
Ex: global variables lifetime will be until the program terminates.
Local variables' lifetime will be limited to the execution time of the function.

“Compilation starts from the first line of the program
Execution starts from the main function, execution USUALLY terminates from the main function.”

VARIABLES DECLARED IN MAIN FUNCTION, WILL HAVE LOCAL SCOPE BUT LONGER LIFETIME, becz EXECUTION STARTS FROM MAIN FUNCTION AND USUALLY ENDS IN MAIN FUNCTION

CONTROL MEANS FIRMWARE WHICH IS A COMBINATION OF HARDWARE AND SOFTWARE.

Program to add two numbers using functions

```
//FUNCTION PROTOTYPE is the function header, but without argument name
// and concluding with ;
void  display1(int, int, int);
void accept(int *, int *);
void add(int *, int, int *);
int main( ) {
                int a, b, sum;
                accept(&a,  &b);  // FUNCTION CALL STATEMENT.
                add(&a, b, &sum);
                display1( a, b, sum);
}
 void  display1( int p, int  q,int s )
{
  printf("%d + %d = %d\n", p, q,s);
}

 void accept( int *p, int  *q)
{
     scanf("%d%d", p, q);
}
//FUNCTION DEFINITION
void  add( int *p, int b, int *sum) // FUNCTION HEADER add function scope
{
    *sum =   **(&p) + b;
}
//FUNCTION DEFINITION = FUNCTION HEADER + FUNCTION BODY
```

If statements modifies the state of the variable from outside the scope of it, and the change is supposed to be seen on Local Variable - use CBR

FUNCTION DECLARATION when it is necessary???????
FD is required when called function is below function call statement.

main ( )
****
// to accept values for a and b
// function accept must read values for a and b
// ‘a’ and ‘b’ variables are present in the **main function scope**.

accept ( )
****
//TASK: accept function is used to accept values for a and b
//TASK of the function decides the return type, name of the function, parameters and //also the statements inside the function.

add( )
*****
//TASK: to add contents of a and b and to store the result in sum
// add function reads values of a and b and updates sum variable
// a and b are passed using CBV and sum using CBR

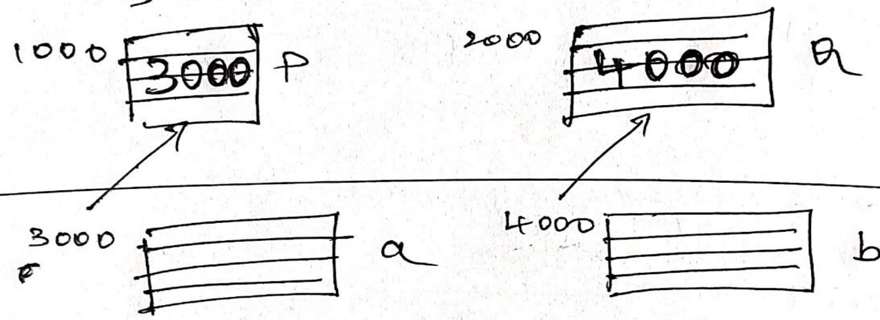
CALL BY VALUE
****
CBV will pass the info/data from one scope to other scope
Arguments in the function header will be variables, to receive the value.

CALL BY REFERENCE
****
CBR will pass the address of a variable from one scope to other scope
Arguments in the function header will be pointers, to receive the address.

accept scope

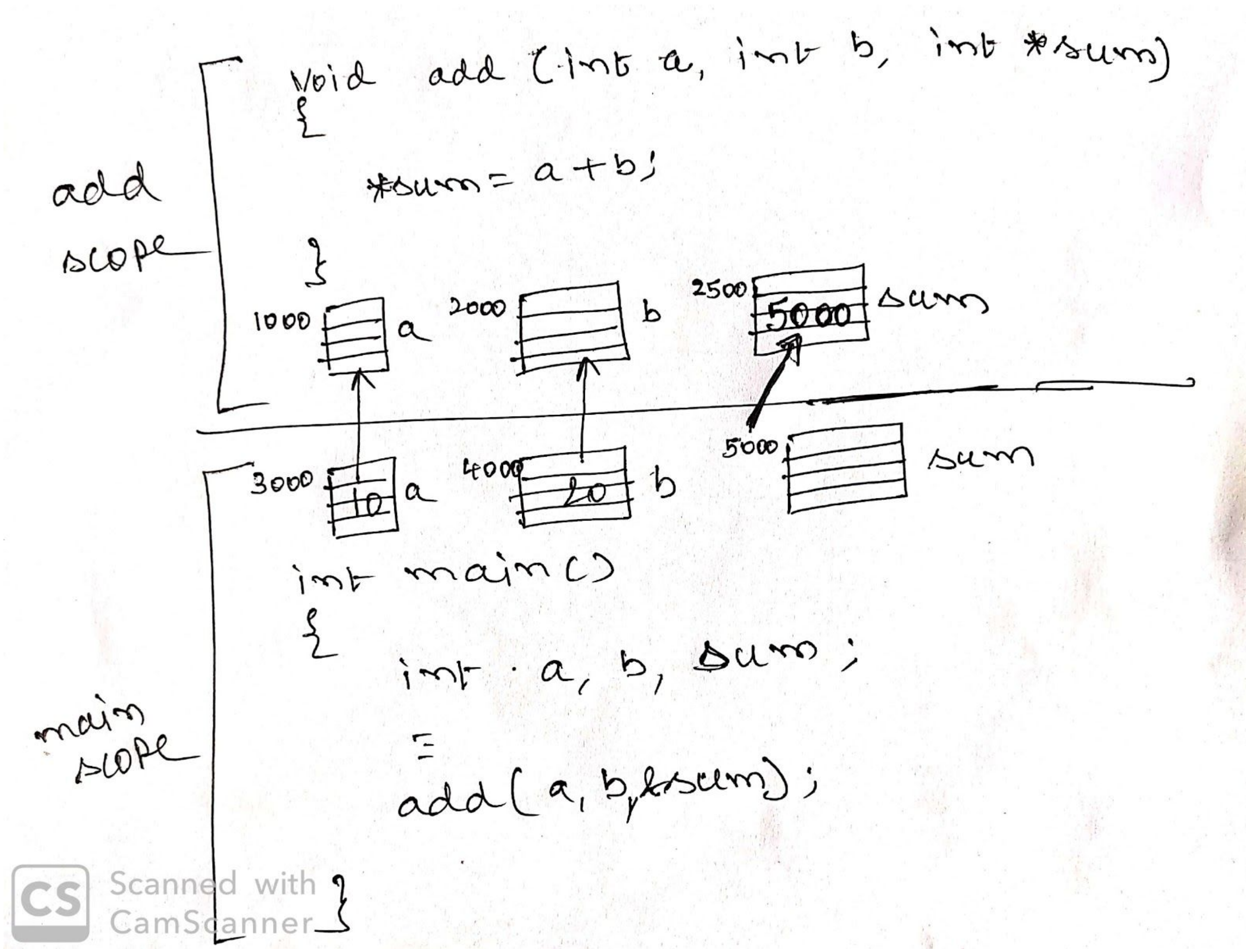
```c
void accept(int *p, int *q)
{
    scanf("%d%d", p, q);

}
```



main scope

```c
int main()
{
    int a, b;sum;
    =
    accept(&a, &b);
```



PROGRAM TO SWAP TWO FLOATING POINT NUMBERS USING FUNCTIONS

```
void accept(float *, float *);
void display(float, float);
void swapping( float *,  float *);
int main( ) {
                float  a, b;
                accept(&a, &b);
                printf("Display before swapping\n");
                display(a, b);
                swapping(&a, &b);
                printf("Display after swapping\n");
                display(a, b);
}
```

```
void   swapping( float *p, float *q) {
        float t;
        t = **&p;       *p = *q;   *q = t;
}
void    display( float p, float q) {
        printf("%f %f\n", p, q);
}
void    accept(   float *p, float *q)
{
        scanf("%f%f",  *&p, q);
}
```

GENERIC POINTER
****

```
void  * z;
printf("%d %d", sizeof(z), sizeof(void *) );
```

int a=10, *p=&a;   float *q;
printf("%d",*p);
P is declared as a int * pointer,
Content of p holds the address where integer value is stored.
***P fetches only 4 bytes of memory and prints it in integer format.**

```
int main( ) {
   int   a;
   void *z = &a;
   printf("Content of generic pointer z is %x\n",z);
   scanf("%d", ((int *)z) );
  printf("%d",         * ((int * )z)   );
  * ((float *)z) =10.2;
  printf("%f",         *((float * )z)   );
  strcpy(z,"rns");   printf("%s\n",z);
}
```

Content of z is considered by compiler as void * and not as int *
So, to convey to the compiler, to consider z content as int * type casting of addresses will be used
Type casting is coded within a ( ) prior to pointer name
And within ( ) int * will be coded

HENCE, PROVED MEMORY IS GENERIC

char *strcpy(char *dest, const char *src);

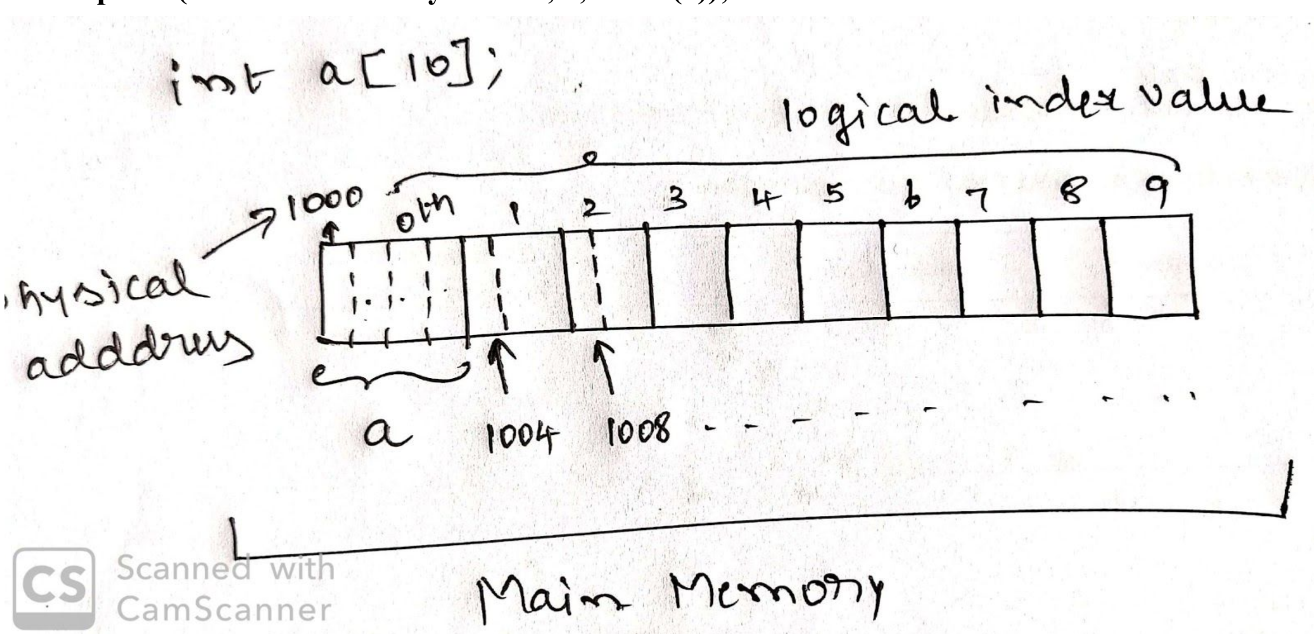
printf(“%x”, strcpy(z, ”rns”));
printf(“%x”, z);

Arrays
****
Array is a collection of similar types of information.
GF: <data-type> <array-name> [size];
int a[10];

‘a’ is an array name.
Array name irrespective of the type provides base/starting address.
Hence array name is a CONSTANT POINTER a++;

***********CONSTANT POINTER, & BASE ADDRESS**********

```
int a[10];
printf("%x   size of array is %d", a, sizeof(a));
```



POINTER ARITHMETIC
****

```
int a[2] = {1,2};
int  *p=0;  //p IS A VARIABLE POINTER & a IS A CONSTANT POINTER
p=a;  printf("Content of p is %x",p);
printf("%d", *p); // output is 1
```

p++;   OR p = p +1; //pointer arithmetic
printf("Content of p after ++ is %x", p);
printf("%d", *p); // output is 2

p--;
printf("Content of p after -- is %x", p);

/*
p++ is converted by the compiler as p = p + 1;
p = p + 1 * sizeof(information pointed by the pointer);   p = p + 1 * sizeof(int)
p=1004
***/**
Important rules for pointer arithmetic
Only integer values must be +ed or -ed from address
Only integer values can be +ed and -ed but cannot be Xed or /ed
P = p *2;   p = p / 2;   CTE

[ ] subscript of index operator

****

int a[2] = {1,2};
printf("%d",a[0]); //a[0] provides value at 0th location of an array
printf("%d", 0[a]); // syntactically correct

GF of [ ]
exp1 [exp2]
Either exp1 or exp2 must provide an address and other expression must yield an integer value.
Further, it is converted by compiler as *(exp1 + exp2), which is nothing but pointer arithmetic. Single set of [ ] will yield one *(dereference operator) after conversion from compiler.
Ex: consider base address of a as 1000
a[0] =   1000[0] = *(1000 + 0 * sizeof(int)) = *(1000)
sizeof(int) is an implicit or invisible multiplication factor provided by the compiler.

int b=90;
printf("%d", ((int *)&b) [0]);   ((int *)&b[0] === b

```
int a[2]={1,2};
a[0]   similar to *(a+0)  printf("%d %d",a[0], *(a+0) );
&a[1] similar to  (a+1)        printf("%x     %x", &a[1] , (a+1) );
```

PARAMETER PASSING TECHNIQUE FOR ARRAY

By default all types of arrays are passed and returned from one scope to another using CBR technique.

Pgm to accept n values for an array and to display it

```
int main( ) {
     int a[10], n;
     void   accept(int *, int *);
     accept(&n, a);
     void display(int, int *);
     display(n, a);
}
void display(int n, int * a) {
  int  i;
  for(i=0;i< *&n; i++)
    printf("%d\n",     *(a+i) );
}


void accept(int *p, int *q ) {
//p holds address of n  and q hold base address of array
     int i;
     scanf("%d", p );
     for(i=0; i<(*p); i++)
        scanf("%d",  &q[i] );   //or    scanf("%d",    (q+i) );
}
```

PGM TO ADD TWO NUMBERS

```
int  main( ) {
  Int a, b, sum;  int *p, *q, *c;
  p=&a; q=&b; c = &sum;
 void accept(int *, int *);
 Void add(int , int, int *);  void display(int, int, int);
  accept(p, q);
  add(*p,*q, c);
  display(*p, *q, *c);
}
void  display(int a, int b, int sum) {
  printf("%d  + %d = %d\n",a, b, sum); }


void  add(int  r, int s, int *t)
{
    *t = r+ s;
}
void accept( int *r, int *s) {
   scanf("%d%d", r,s);
}
```

ARRAY INITIALIZATION

INITIALIZE means many values can be initialized   Ex: int i=10;

ASSIGNMENT only one value can be assigned.   Ex: int i;   i=10;

int a[10] = {1, 2, 3};

A[0] will be initialized with value 1

A[1] will be 2

A[2] …… 3

A[3] to a[9] will be automatically initialized to zero value, and this happens only for numerical arrays.

float b[3] = {10.2, 3.2}

int a[10];
a = {1,2,3};   //CTE

int a[ ] = { 1,2,3,4}; printf("%d",sizeof(a));
When initializers are present, the size of an array can be omitted.

CHAR ARRAY INITIALIZATION
Char array's can be initialized with character values or string value
Character values are the one which are enclosed in single quotes '
String values are the one which are enclosed in "

char a[ ] = { 'r','n','s','i','t' };
char b[ ] = { 'r','n','s','i','t','\0' }; // b can be used as string also

If a string is stored in a character array (i.e last char is '\0'value), then it can be accessed in character manner.

If a char array is storing only characters (i.e no null value at the last), then it cannot be used as a string.

char z[]={"zoo"}; printf("%d", sizeof(z));
char y[30];
y = {"zoo"}; // CTE
strcpy(y,"zoo"); // "zoo" is a literal string constant

DYNAMIC MEMORY ALLOCATION
Memory allocation for an information or for a variable can be done in 2 ways
1. static/automatic memory allocation
2. Dynamic memory allocation

1. AMA /SMA will be done by OS just before execution starts
2. DMA will be done during the execution of the program.
DMA in c is performed using 3 functions
1. malloc( ) 2. calloc( ) 3. realloc( )
All these are built-in functions

```
typedef int INTEGER; //typedef is used to create an alias for data type
INTEGER a;
```

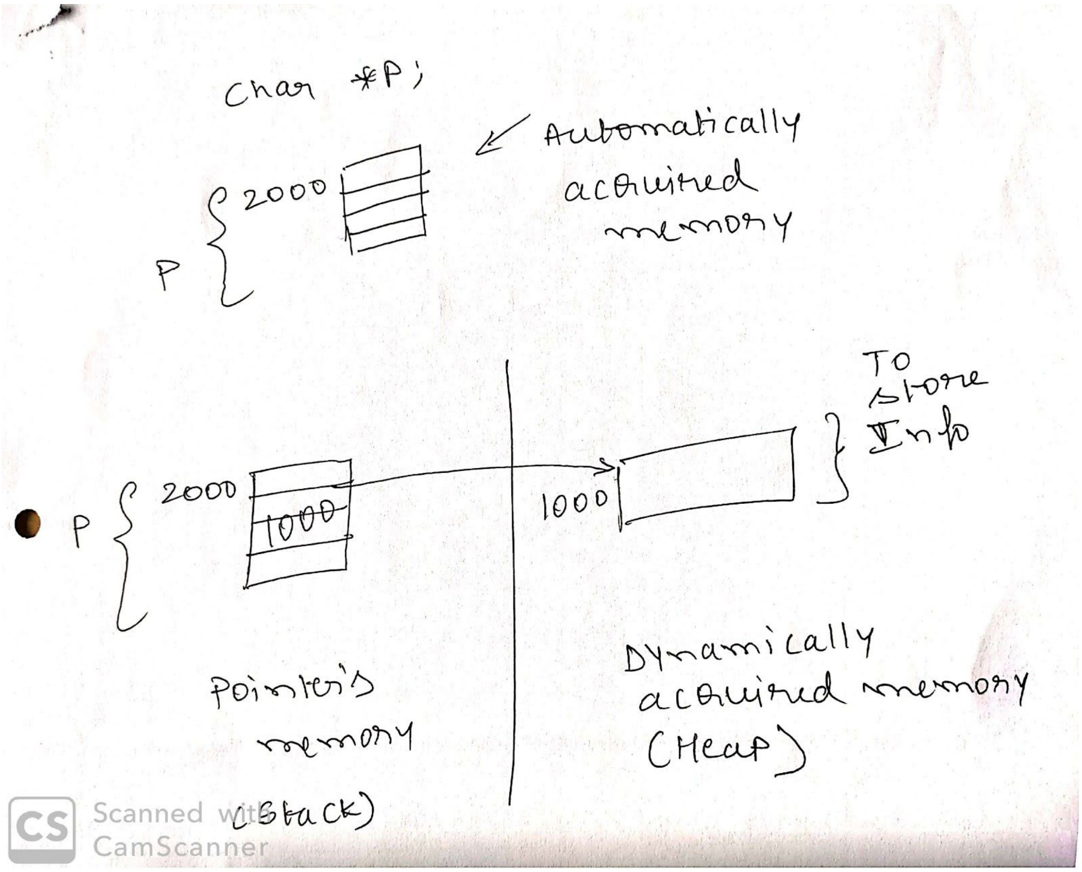
```
#include <stdlib.h>
void * malloc(size_t size); //size_t is a synonym for unsigned int
```

```
int *p;
p = (int *) malloc(4);
// malloc will try to acquire 4 consecutive bytes of MM location.
//after acquiring 4 bytes of memory 1st bytes address will be returned
//by malloc.
```

```
float *q;
q = (float *) malloc( sizeof(float) );
```

WHENEVER DMA IS USED TO ALLOCATE MEMORY TO STORE VALUES, C PROGRAMMER HAS TO USE **UNNAMED/POINTER ACCESS** TECHNIQUE TO ACCESS THE VALUE.

```
# include <stdlib.h>
int main( ) {
   auto int *p=0;
   p  = (int *) malloc( sizeof(int) );
   *p =10;  printf("%d",*p);    free(p); }
```

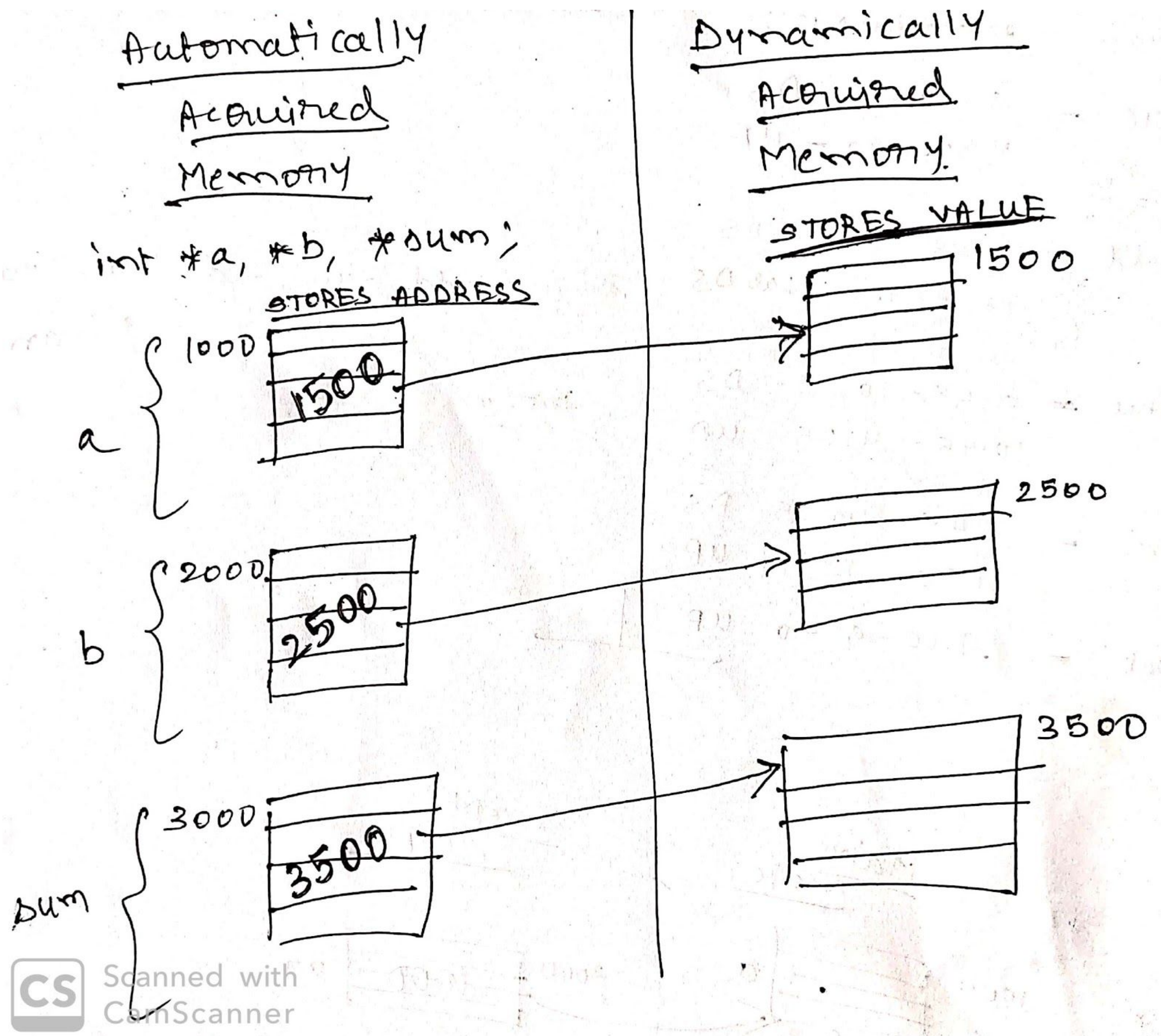


Dynamically acquired memory must be released explicitly by the programmer before the program completes its execution.

void free(void *ptr);
Resubmitting/Release of main memory to OS for further usage.

WAP TO ADD 2 NOS USING DMA

```
int main( ) {
   int *a=0, *b=0, *sum=0;  //null pointers
   a=(int *) malloc(sizeof(int));
   b=(int *) malloc(sizeof(int));
   scanf("%d%d", a, b);
   printf("Accepted values are%d %d\n", *a, *b);
   sum=(int *) malloc(sizeof(int));
   *sum = *a + *b;
   printf("Result after addition is %d\n",*sum);
   free(a);  free(b);  free(sum);
}
```

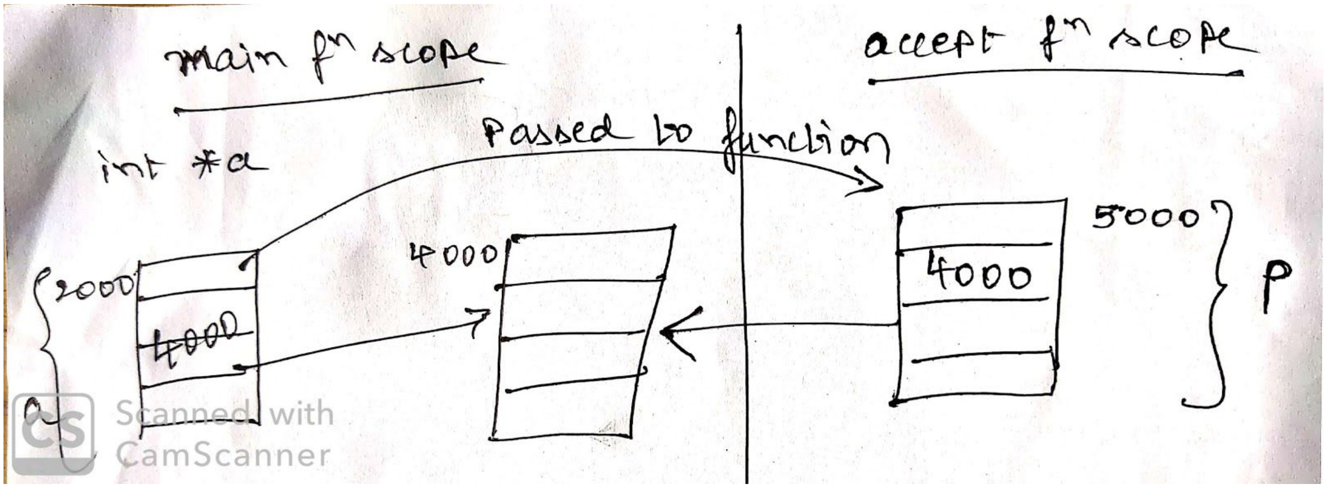


WAP to add two numbers using modules and DMA technique.

```
//TASK of allocate ( )
//To acquire DM of requested bytes and return the address.
int * allocate(int);
void accept(int *, int *);
void add(int , int, int *);
void display(int, int, int);
int main( ) {
   int *a=0, *b=0, *sum=0;  //null pointers
     a = allocate(sizeof(int)); // OR     a = (int *)  malloc(sizeof(int));
     b= allocate(sizeof(int));
      printf("%x %x\n", a, b);
       accept(a, b);
     sum=allocate(sizeof(int));
       add(*a, *b, sum);   // OR *sum = *a + *b;
     display(*a, *b, *sum);
   free(a); free(b);  free(sum);
   printf("%x  %x  %x\n",a, b, sum);
}
/*free( ) WILL NOT CHANGE THE CONTENT OF POINTER TO ZERO
OR SOME OTHER ADDRESS.  free( ) IS JUST AN INDICATION TO OS
TO USE THE LOCATIONS POINTED BY a, b and sum FOR SOME
OTHER PURPOSE
*/
void display(int p, int q, int r) {
        printf("%d + %d = %d",p,q,r);
}
void  add(int p, int q, int *r) {
    *r = p + q;
}
```

```
 void   accept(int *p, int *q) {
   printf("%x %x\n",p, q);
   scanf("%d%d",p,q);
}

 int *   allocate( int sz) {
    int *p = (int *) malloc(sz);
//p is a local pointer, whose content will be lost as soon as control leaves
//allocate scope.  So we update the content of p to the pointer a which is
//present in main scope.
    return p;
   }
```

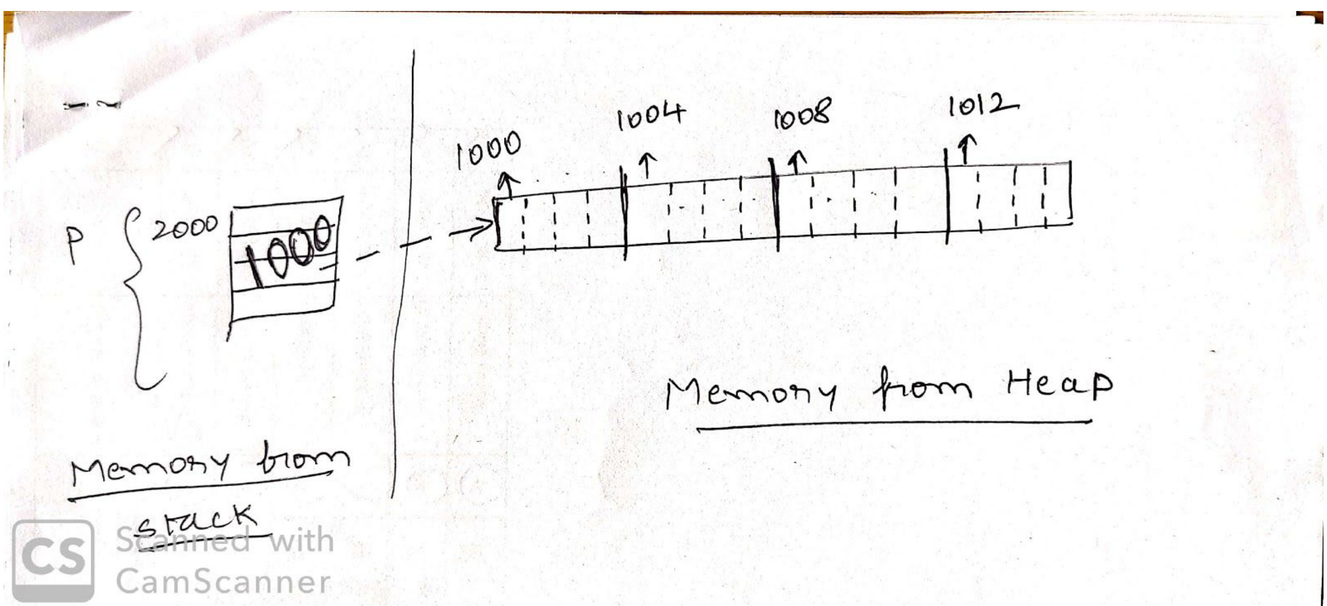


DMA FOR ARRAY'S

int *p =0;

p = (int *) malloc(sizeof(int) * 4);

printf("%x \n ". p); // p prints base address



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

printf("%x\n", p +i);

```
for(i=0;i<4;i++)
  scanf("%d", p +i  );

for(i=0;i<4;i++)
  printf("%x %d\n", p+i,  *(p +i)  );
```

WHENEVER DMA IS USED TO ALLOCATE MEMORY FOR STORING INFORMATION, initialization IS NOT POSSIBLE.

WAP TO FIND THE FIRST AND SECOND LARGEST VALUES IN AN INTEGER ARRAY, USING DMA AND MODULES

```
void  accept(int, int *);
void display(int, int *);
void ls(int, int *, int * int *);
int   main( ) {
    int  n, *a=0,larg, slarg;
    scanf("%d", &n);
    a  = (int *) malloc(sizeof(int) * n);
    accept(n,a);     display(n,a);
    printf("Largest: %d  Second Largest: %d", larg, slarg);
}
```

```
void   ls(int n, int *a, int *l, int *sl) {
    int i;
   *l=*sl=a[0];
    for(i=1;i<n;i++)
       if( a[i] > *l) {
          *sl = *l;  *l=a[i];
      }
       else
       if (a[i] > *sl) *sl= a[i];
}
void   display(int n, int *a) {
      int i;
      for(i=0;i<n;i++)
          printf("%d\n",*(a+i));     }

 void   accept(int n, int *a) {
     int i;
     for(i=0;i<n;i++)
       scanf("%d",   a+i );   //&a[i]     }
```

Data types : Categorized into 2 types

1. Built-in/Primitive data types   ex: int, char, float
2. User-defined data types   ex: struct, union, enum

Structure

Structure is a collection of different data types, which are logically related.

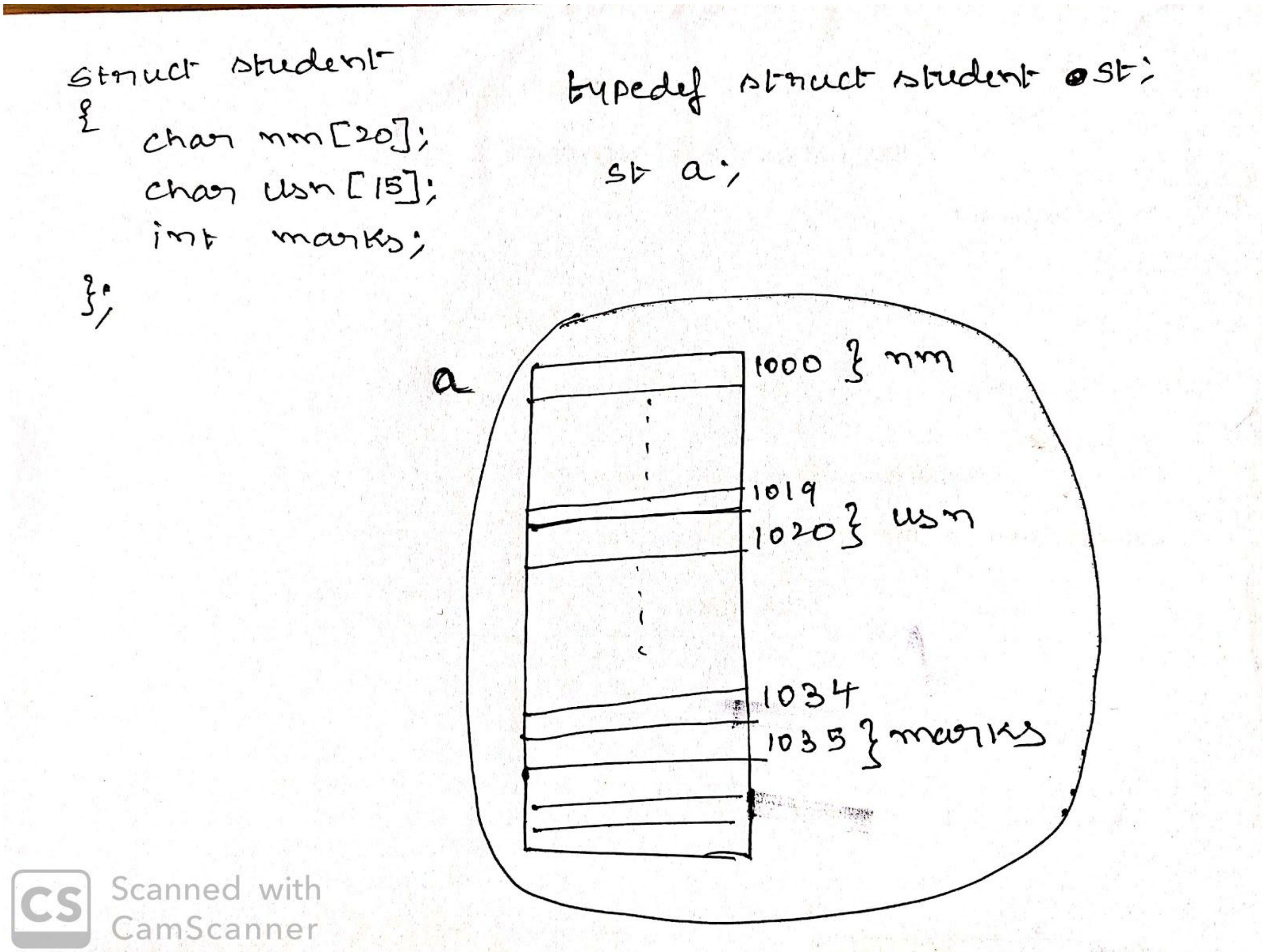
G.F:

```
struct <structure type>
{
    members
};
```

Ex:

```
struct student  // student is a data type
{
   char nm[20];
   char usn[15];
   int marks;
};
```

Struct student is a data type and **no memory is allocated to the student, until memory is acquired to store information of type struct student.**



struct student // student is a data type

{

char nm[20];

char usn[15];

int marks;

};   typedef struct student st;

st a; // a is a variable of type struct student

'a' designates nm, usn and marks.

// prints the total amount of memory acquired by a

printf("%d", sizeof(a)); //sizeof(st);

// prints starting address of structure.

printf("Starting address of a %x", &a);

```
int a;   char n[20];
scanf("%d",&a);
printf("%d",a);

scanf("%s", n);
printf("%s", n);
```

Contents of 'a' can be accessed as a whole or in member wise manner.
In order to access different members in structure two operators are used in C.

1. "." direct member selector
   must be used when a **non-address or an identifier of a structure** (i.e a) is used to select a member
   printf("%s ", **a.nm**);
   'a' is a non-addressable or an identifier which represents a struct student type.

2. "->" indirect member selector
   must be used when an **address of a structure** is used to select a member.
   printf("%s", **(&a)->nm**);
   &a provides the first byte address of the structure
   First byte address is also the address of the first member in structure.

ALIASING a struct type
typedef is used to create an alias for a data type
GF: typedef <actual-data-type> <alias-name>;
EX: **typedef struct student st;**

HOW TO CHOOSE MEMBER AND CHOOSE OPERATOR
st a;
ACCEPTING VALUE FOR nm field in structure
1. Variable of type struct student is a, and ‘a’ represents non-address entity hence ‘.’ must be used to choose any member not only nm.
scanf(“%s”, a.

2. Upon choosing nm, the type of nm must be considered, which is an array name. Array name provides a base address straight away, which is essential in a scanf statement.
scanf(“%s”, a.nm);

ACCEPTING VALUE FOR marks
1. ……
scanf(“%d”, a.
2. Upon choosing marks, the type of marks is a numeric variable, referring to a numeric variable name provides value, but scanf demands address. Address of ‘marks’ must be extracted upon choosing it from ‘a’.
scanf(“%d”, &(a.marks)); address of marks

USING -> OPERATOR TO CHOOSE MEMBER nm

st a;

1. ‘a’ is a variable of type st, instead of considering a, we choose &a, which designates the address of a. If the address of a is considered, then -> operator must be used to choose a member.

scanf(“%s”, (&a)->nm);

Further, & is not needed to extract address from nm, because nm is array name, which straight away provides base address.

USING -> OPERATOR TO CHOOSE MEMBER for marks

1. …..

scanf(“%d”, (&a)->marks);

2. Above statement will generate RUN-TIME ERROR when executed because the address of marks has to be explicitly accessed and passed to scanf.

scanf(“%d”, &((&a)->marks));

ACCEPTING VALUES FOR MEMBERS IN STRUCTURE

st a;

scanf(“%s%s%d”, a.nm, a.usn, &(a.marks));

PRINTING VALUES OF THE MEMBERS OF A STRUCTURE

printf(“%s %s %d\n”, a.nm, a.usn, a.marks);

PGM TO ACCEPT STUDENT TYPE AND TO DISPLAY THE SAME.

```
int main( ) {
   struct student
   {
      char nm[20];
      char usn[15];
      int marks;
    };
   typedef struct student st;
   st  a;

   scanf("%s%s%d",    );
   printf("%s %s %d\n",  );
}
```

PGM TO ACCEPT VALUES FOR STUDENT TYPE OF VARIABLE AND DISPLAY THE SAME USING MODULES.

IN THIS PROGRAM STRUCT DECLARATION WILL BE DONE OUTSIDE main( ) FUNCTION BECZ, THE MODULES USED IN THE PROGRAM MUST BE ABLE TO ACCESS STRUCT DECLARATION.

```
struct student {
   char nm[20];
   char usn[15];
   int marks;
};
typedef struct student st;
void display(st );
```

```
void accept(st *);
int main( ) {
      st a;
      accept(&a);
      display(a);
}
void display(st p) {
   printf("%s %s %d\n", p.nm, p.usn, p.marks);
}
void accept(st *p) {
 // p holds on to the address of a
// p == &a    and   *p  == a
scanf("%s%s%d", p->nm, (*p).usn, &(p->marks));
}
```

PGM TO ACCEPT A VALUES OF TYPE STRUCT STUDENT AND DISPLAY IT USING DMA

```
struct student {
   char nm[20];
   char usn[15];
   int marks;
};
typedef struct student st;
int main( ) {
    st * p =0;
   printf("size of st * %d", sizeof(p) );
   // Dynamic memory will be acquired in order to store st type of info.
   p = (st *) malloc(sizeof(st));
   scanf("%s%s%d", (p->nm), p->usn, &(p->marks) );
   printf("Name: %s\nUSN: %s\nMarks: %d",   (p->nm), p->usn,
```

```
                                                p->marks);
   free(p);
}

INITIALIZE VARIABLE OF TYPE st
st  a= {“sushruta”, ”1RNEE101”, 89};
st  b=a;
printf(“%s %s %d\n”,   a.nm, (&a)->usn, a.marks      );
printf(“%s %s %d\n”,   b.nm, (&b)->usn, b.marks      );
*****************
char a[10] = {“abcd”};
char b[10];   strcpy( &b[0], a+2);     printf(“%s”, b);
&b[0]
b[0]  == *(b+0)
&b[0] ⇒  (b)

struct ar{
   char a[10];
};
typedef struct ar  arr;

arr  p={“abcd”},q;
printf(“%s”, p.a);
q=p; //Assignment statement
printf(“%s”, (&q)->a);
```

ASSIGNING VALUE FOR A VARIABLE OF TYPE st

```
struct student {
   char nm[20];
   char usn[15];
   int marks[2];
};
typedef struct student st;
st a; //memory will be acquired for a in automatic/static manner
strcpy(a.nm,"sushruta");
strcpy(a.usn ,"1RNEE101");
a.marks[0] = 90;   a.marks[1]=91;
printf("%s %s %d %d\n", a.nm, a.usn, a.marks[0], a.marks[1]  );
```

ASSIGNING VALUE FOR A VARIABLE OF TYPE st, WHOSE MEMORY IS ALLOCATED IN DMA MANNER.

```
struct student {
   char nm[20];
   char usn[15];
   int marks;
};
typedef struct student st;

st  *p = 0;
p = (st *) malloc(sizeof(st) );
strcpy(p->nm , "sushruta");
strcpy(p ->usn,  "1rnee101");
p->marks = 89;
```

PGM TO ACCEPT ‘N’ STUDENT INFORMATION AND TO PRINT IT.

```
struct student {
        char nm[20],usn[15];
        int marks;
};
typedef struct student st;
int main( ) {
    st arr[10];
    int n,i;
    scanf(“%d”, &n);
/*  arr is the name of array - provides base address - also it is a constant
                                                                  pointer
    arr[0] or *(arr+0) provides the full information at 0th location
    &arr[0] or (arr+0) provides the address of 0th location
*/
    for(i=0;i<n;i++)
      scanf(“%s%s%d”,   arr[i].nm,arr[i].usn, &(arr[i].marks)   );

    for(i=0;i<n;i++)
       printf(“%s %s %d”,  (&arr[i])->nm, arr[i].usn, arr[i].marks );
}
```

PGM TO ACCEPT 'N' STUDENT INFORMATION AND TO PRINT IT USING MODULES AND TO PRINT THE STUDENT NAME WITH HIGHEST MARKS.

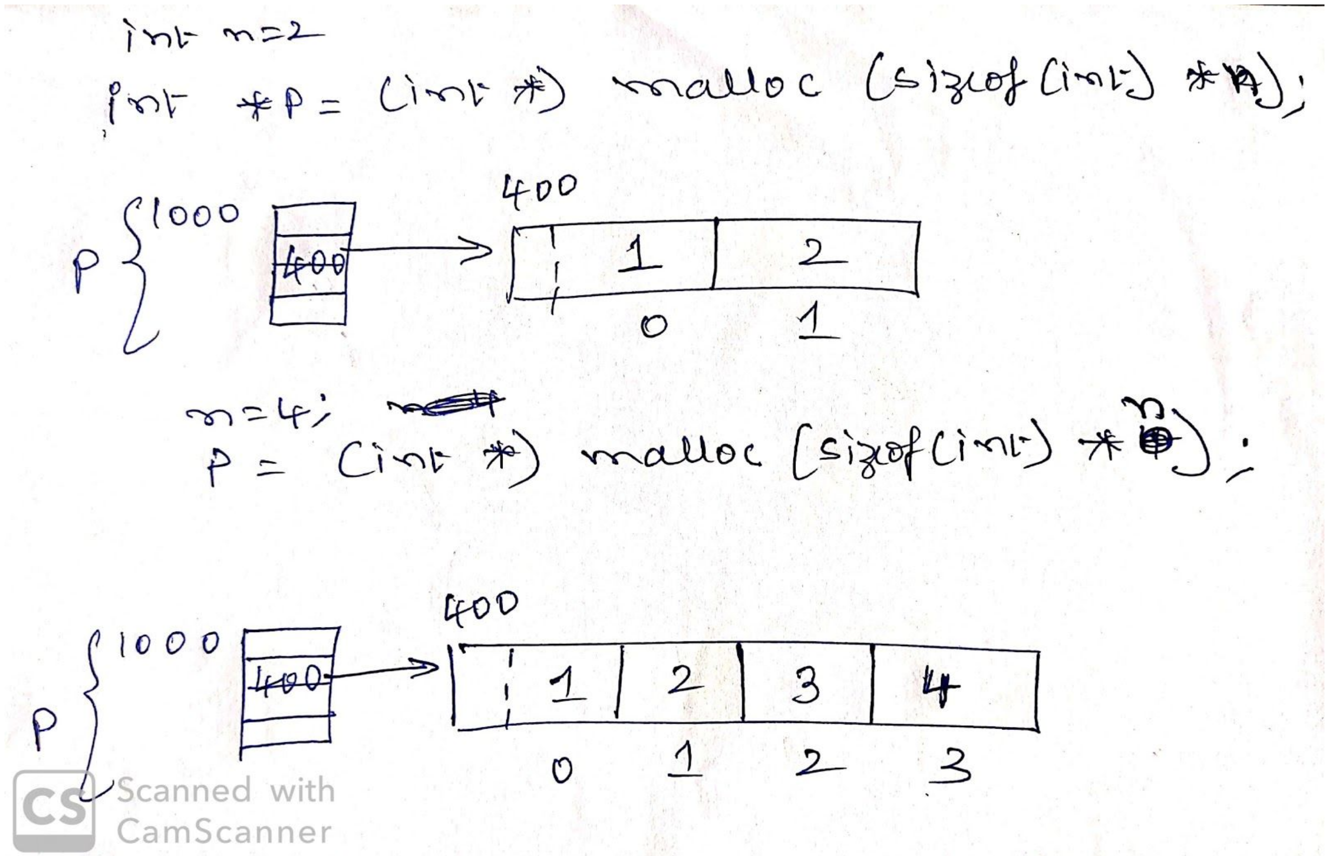
```
struct student {
        char nm[20],usn[15];
        int marks;
};
typedef struct student st;
void accept(int *, st *);
int main( ) {
     st arr[10];
     int n;
     accept(&n,arr);
}
void  accept(int *p, st * arr) {
//    arr in accept scope is a variable pointer
    int i;
    scanf("%d",p);
    for(i=0;i<(*p);i++)  // *p === n
       scanf("%s%s%d", arr[i].nm, arr[i].usn, &(arr[i].marks) );
}
```

calloc( )
****
void *calloc(size_t nmemb, size_t size);
calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory.
The memory is set to zero.
calloc() returns a unique pointer value that can later be successfully passed to free().

```
int *p = (int *) calloc(1,sizeof(int));
printf("%d", *p);

char *q = (char *) calloc(3,sizeof(char));
printf("%s", q);
//output will be empty, means 3 bytes are initialized to ASCII value 0
```

realloc( )
****
void *realloc(void *ptr, size_t size);



The realloc() function changes the size of the memory block pointed to by ptr to size bytes.

The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.

If the new size is larger than the old size, the added memory will not be initialized.

If ptr is NULL, then the call is equivalent to malloc(size), for all values of size;

If size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).

If ptr is not NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().