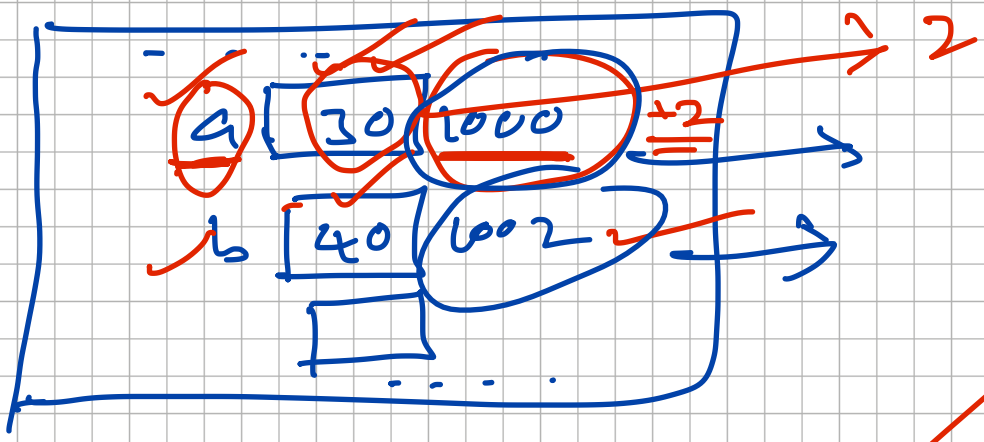
Pointer Concepts:

```
int a=30;   — 1
int b=40;   — 2
```




```
printf("%d",a);  = 30
```

Pointers:

Program to print the value using variables and their addresses.

```
#include <stdio.h>
void main()
{
    int a=20; int b=40;
    printf("value of a = %d", a);
    printf("value of b = %d", b);
```

↳ Accessing the data using variable printing

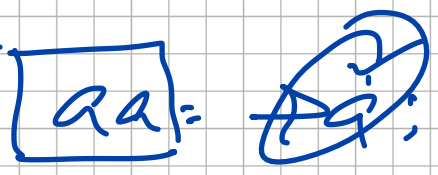
```
    printf("Address of a = %d", &a);
    printf("Address of b = %d", &b);
```

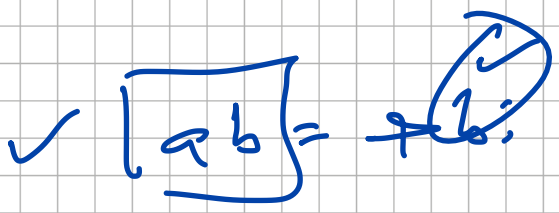
↳ Accessing the address of variables.

```
    printf("value of a = %d", *&a);
    printf("value of b = %d", *&b);
}
```

*&a

↳ Accessing the data using de-referencing operator.

✓  the variable aa contains the address of a variable ?

✓ 

Pointers / Pointer Variables.

Pointer: A variable that contains the address of another variable or address of a memory location is called a Pointer. A pointer is also called as a pointer variable.

Instructions:

1. Declare a data variable

   Ex: int a, b;

2. Declare a pointer variable

   Ex: int *p;

3. Initialize a pointer variable

   Ex: p = &a;

4. Access data by using pointer variable

   Ex: printf("%d", *p);

Pointer declaration

type *variable;

type → data type of the pointer variable

* → name given to the pointer variable

Indicates that the identifier is a pointer variable

→ data type

Ex:

If a variable P contains address of int variable, it is declared as

int *P;

If a variable x contains address of float variable, then it is declared as float *x

If a variable y contains address of char variable then

char *y;

int *P; int* P; int * P;

If a variable fp contains the address of a FILE which, its declaration is: FILE *fp;

AMOUNT → user-defined data types.

```
typedef int AMOUNT;
```

AMOUNT *p;

Consider:

```
struct student
{
    char name[20];
    int marks;
    float avg;
};
struct student *b;
```

Consider the following statement

```
int *pa, pb, pc;
```

✓

Here only pa is the pointer variable

pb & pc are the ordinary variables.

```
int *Pa, Pb, Pc;
```

✓

↓

```
int *pa;
int pb, pc;
```

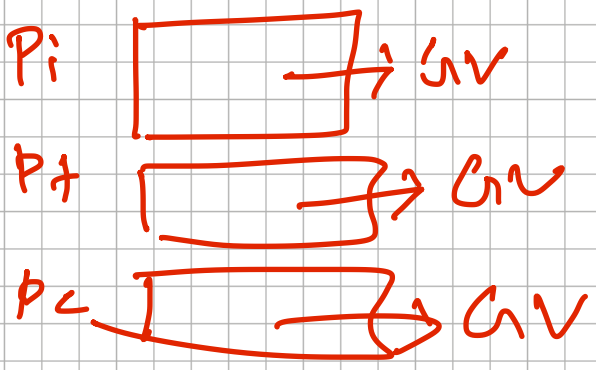
→ ✓

||

```
int *pa, pb, pc;
```

Consider following declarations and assume that all are local variables:

```
int   *pi;
float *pf;
char  *pc;
```



The local variables are not initialized by the compiler during the execution. This is because, the local variables are created and only during the execution time.

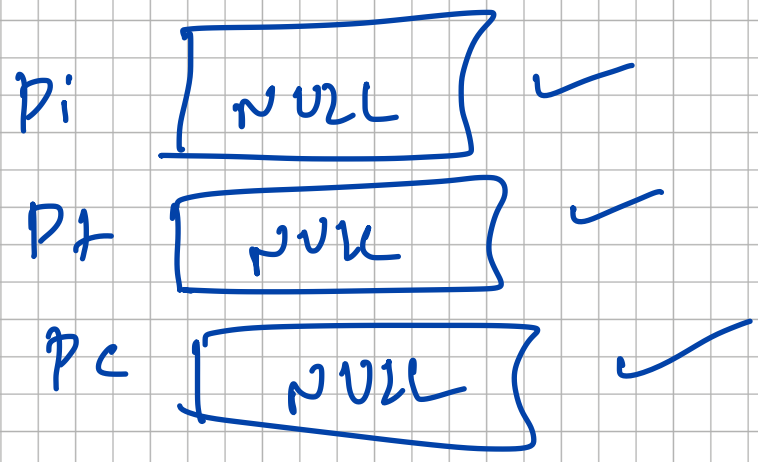
The same will be applicable for pointers also.

A pointer variable which does not contain a valid address is called

Dangling pointer.

Consider following declaration all are global variables.

```
int  *pi;
float *pf;
char *pc;
```



NULL Pointer: A NULL pointer is defined a special pointer value that points to nowhere in the memory.

If it is too early in the code to assign a value to the pointer, then it is better to assign NULL (10 or 0) to the pointer.

Consider

```
#include<stdio.h>
int *p = NULL;
```

This indicates that the pointer var. p does not point to any part of the memory.

Note: A pointer variable must be initialized. If it is too easy to initialize a pointer variable, then it is better to initialize all the pointer variables to null in the beginning of the code.

Declaring a data variable, pointer variable and initializing pointer variable

```
int n;
int *pn;
pn = &n;
```

(OR)

```
int x;
int *px = &x;
```

(OR)

```
int x, *px = &x;
```

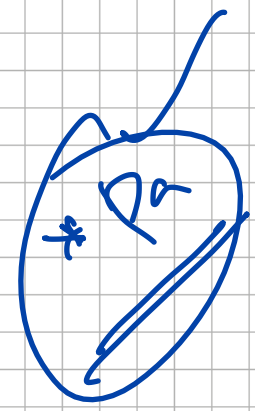
Consider:

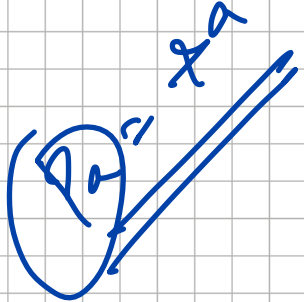
```
int p, *ip;
float d, f;
```

ip = p; ⟶ ERROR

ip = &d ⟶ ERROR

ip = &p; ⟶ ✓



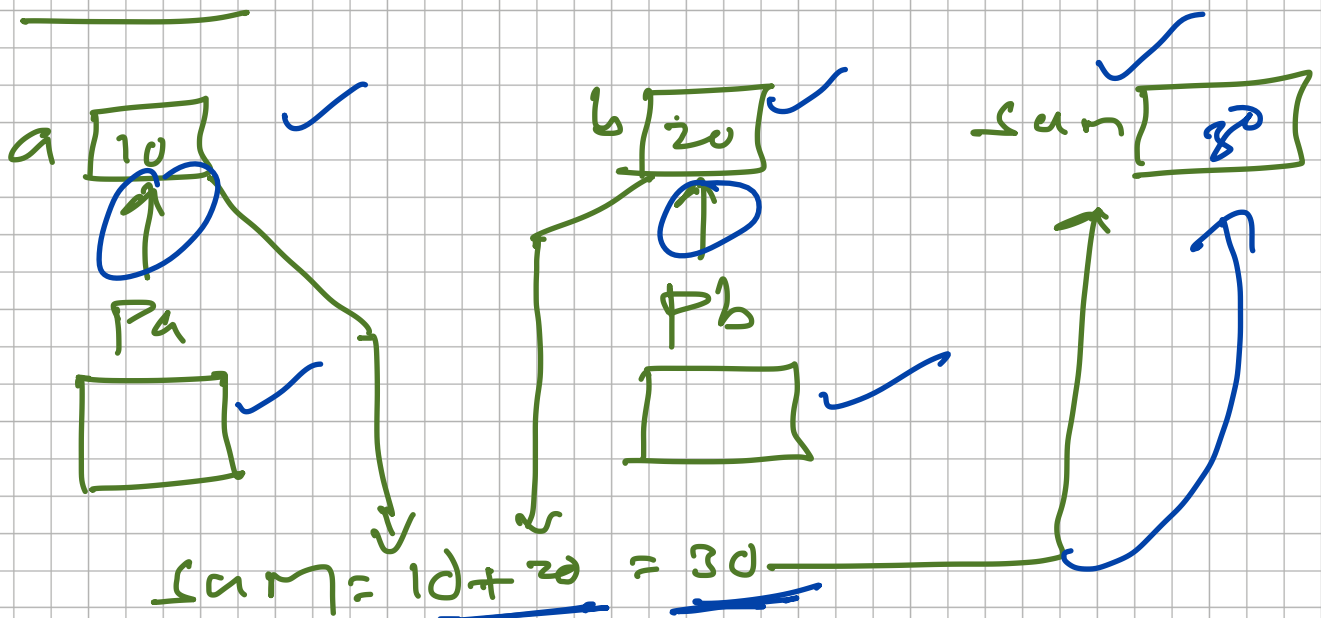


Program to add two numbers using pointers:

```
#include<stdio.h>
void main()
{
    int a=10, b=20, sum;
    int *pa, *pb;
    pa = &a;
    pb = &b;
    sum = *pa + *pb;
    printf("sum = %d", sum);
}
```

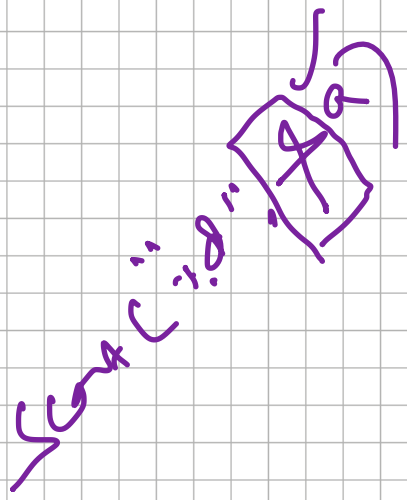
sum = a+b

Tracing:



To read two numbers and add two numbers using pointers.

```
#include <stdio.h>
void main()
{
    int a, b, sum;
    int *pa, *pb;
    pa = &a;
    pb = &b;
    scanf("%d %d", &a, &b);
    sum = *pa + *pb;
    printf("sum = %d", sum);
}
```



scanf("%d %d", pa, pb);

scanf("%d %d", &a, &b);

Pointers are flexible

The pointers are very flexible and can be used in variety of situations:

→ A pointer can point to different memory locations. — [1] ✓

→ Two or more pointers can point to same memory location. — [2]

→ Altering functional arguments using pointers. ✓

→ Functions returning pointers. ✓

→ Pointers to pointers.

→ Arrays and Pointers. ✓

→ Pointer can point to a single dimensional array.

→ Arrays of Pointers.

→ Pointers can point to a function. ✓

[1]:

```
int x=10, y=20, z=30;
int *p;
p=&x;
printf("%d", *p);
p=&y;
printf("%d", *p);
p=&z;
printf("%d", *p);
```

10

20

30

[2]:

consider

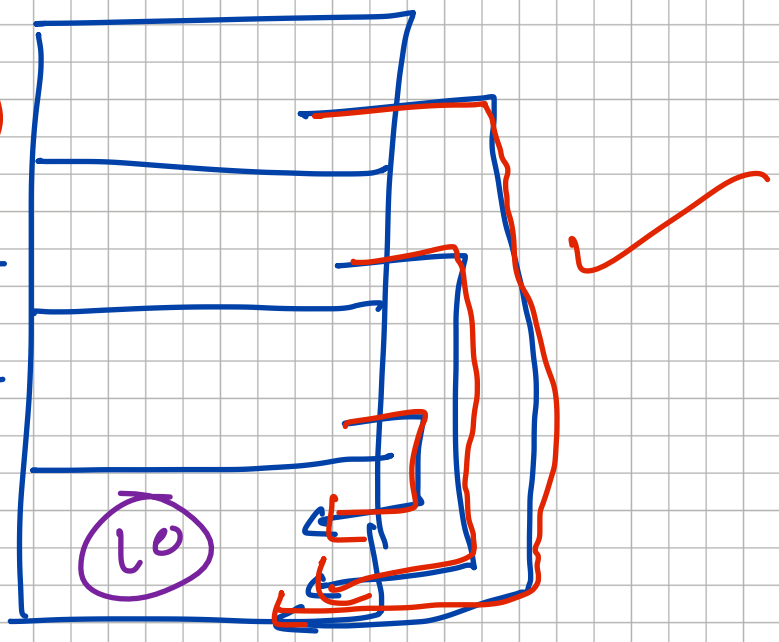
```
int *p;
int *q;
int *r;
int x=10;
```

```
p = &n;
z = &n;
r = &n;
```

p 5000

z 5004

r 5006




```
printf("&p = %u, p = %u, *p = %d", &p, p, *p);
```

5000 5006 10

```
printf("&z = %u, z = %u, *z = %d", &z, z, *z);
```

5002 5006 10

```
printf("%u, &r = %u, r = %u, *r = %d", &r, r, *r);
```

5004 5006 10

Pointers and Functions:

Program to print both maximum and minimum of two numbers.

```
#include<stdio.h>
void main()
{
    int m,n, big, small;
    printf("Enter two values");
    scanf("%d %d", &m, &n);
    maxmin(m,n, &big, &small);
    printf("Largest = %d", big);
    printf("Smallest = %d", small);
}
void maxmin(int a, int b, int *max,
                          int *min)
{   if(a>b)
    {   *max = a, *min = b;
        return;
```

*max
max

```
    }
    *max = b, *min = a;
}
```

```
maxmin(n, n, &big, &small);

maxmin(int n, int n, int *max,
                     int *min)
```

max = &big

min = &small

*max = a

mem

Function Returning pointers:

```
void main()
{
    int x, y, *big;
    printf("Enter the value of x & y");
    scanf("%d %d", &x, &y);
    big = largest(&x, &y);
    printf("max(%d, %d) is : %d", x, y, *big);
}
int largest(int *a, int *b)
{
    if(*a > *b)
        return a;
    else
        return b;
}
```

big = largest(&x, &y);

↳ int largest(int *a, int *b)

a = &x, b = &y

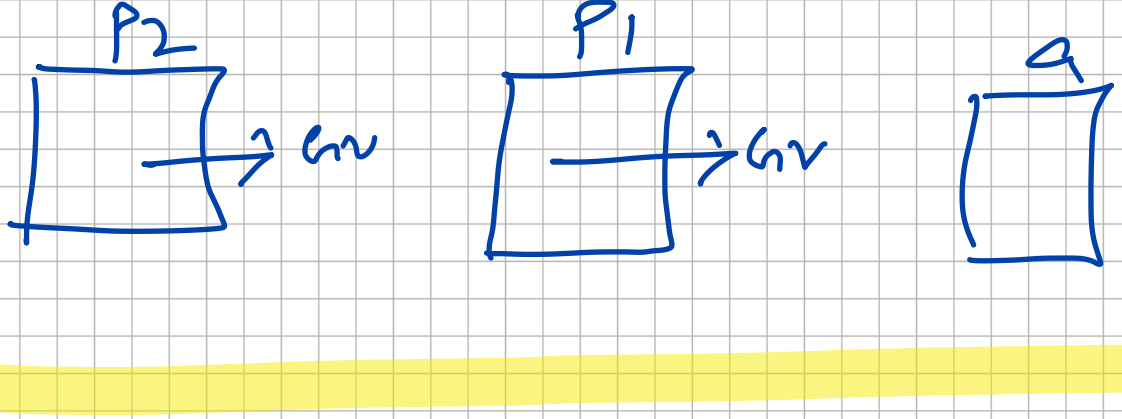
*a *b

Pointers to pointers:

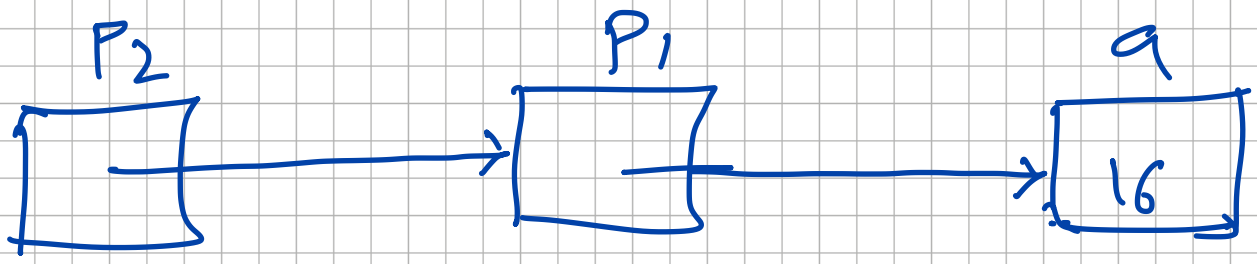
A variable which contains the address of a pointer variable is called pointers to pointer.

Ex:

```
int a;
int *p1;
int **p2;
```




```
a = 16;
p1 = &a;
p2 = &p1;
```



printf("%d", a);
a

printf("%d", *P1);
a = *P1 =

printf("%d", **P2);
a = *P2 = 16

Consider:

```
int a=5;
int b=7;
int *p= &a;
int *q = &b;
```

Find the value of each expression:

```
++a;
++(*p);
```

$$\left.\begin{array}{l} --(*a); \\ --b; \end{array}\right\} \Rightarrow a=7;\ b=5;$$

int a=5;

++a $\Rightarrow$ a = 6

++(*p) $\Rightarrow$ a = 7

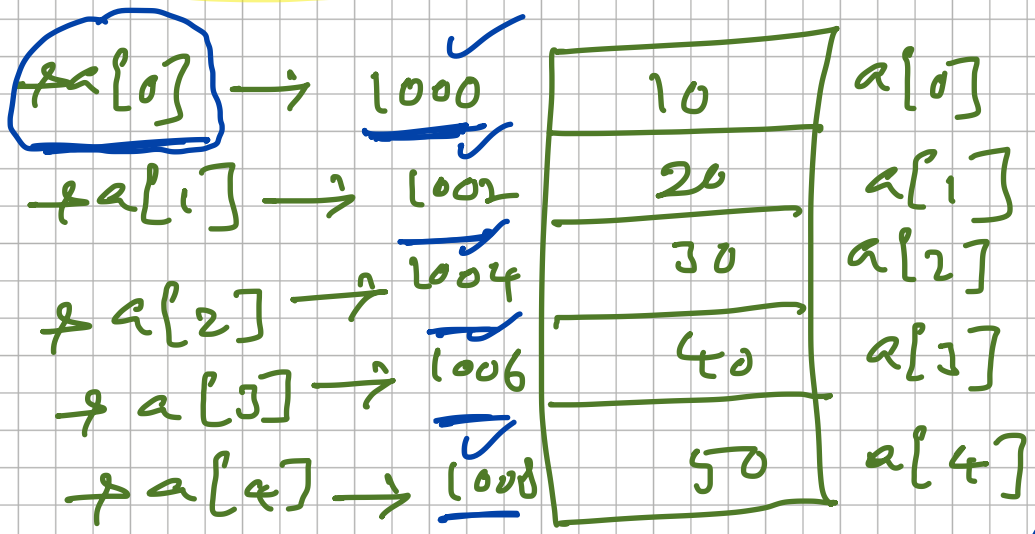
--(*a) $\Rightarrow$ b = b-1
= 7-1 = 6

--b $\Rightarrow$ 5

Arrays and Pointers:

Consider the following declaration:

```
int a[5] = {10, 20, 30, 40, 50};
```



Note: &a[0] & a are same

&a[i] == &a[0]

Pointer constant:

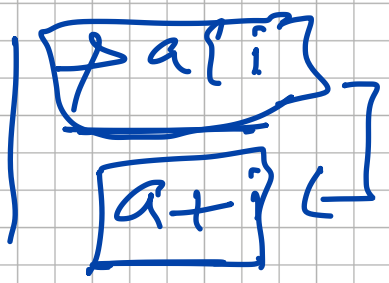
The address of the memory location 0100 stored in a can't be changed. So, even though a contains an address, since its value can't be changed, we call a as pointer constant.

Consider the Program

```
#include<stdio.h>
void main()
{
    int a[5]={10,20,30,40,50};
    Printf("%u,%u,%u",&a[0],a,a+0);
}
```

o/p: 0100, 0100, 0100

x x x



How to access the address of each element →

address operator will index

base address will index ✓

&a[0] — 1000 ✓ (a+0) — 1000 ✓

&a[1] ⟷ (a+1)

&a[2] ⟷ (a+2)

&a[3] ⟷ (a+3)

&a[4] — 1008 ✓ ⟷ (a+4) — 1008 ✓

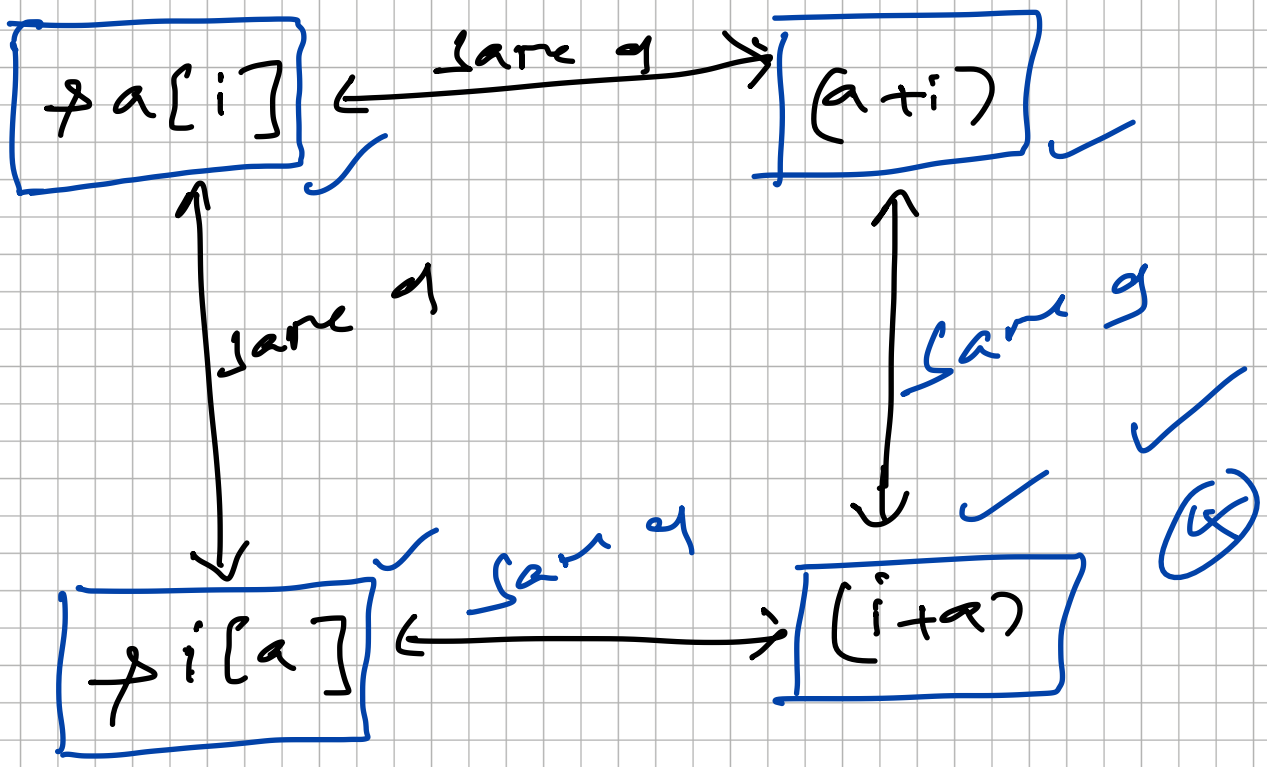
In general &a[i] ⟷ (a+i)

for i=0 to 4

i.e., i=0 to n-1

&a[i] a+i

Note: The various ways of accessing the address of the item in an array a.



∴ The address of a[i] is:

&a[i] or &i[a] or (a+i) or (i+a).

The contents of a[i] is:

*&a[i], *&i[a], *(a+i), *(i+a)

a[i], i[a]

Consider the following program

```
void main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int i=3;
    printf("%d %d %d %d %d %d", *(a+i),
        a[i], *(a+i), *(i+a),
        i[a], *(i+[a]));
}
```

o/p: 40, 40, 40, 40, 40, 40

Program to compute largest element & it's position:

```
#include <stdio.h>
void main()
{
    int a[10], n, i, big, pos;
```

```
printf("Enter the no. of elements");
scanf("%d", &n);
printf("Enter the elements");
for(i=0; i<n; i++)
    scanf("%d", &a[i]);

big = a[0];
pos = 0;
for(i=1; i<n; i++)
    if(big < a[i])
    {
        big = a[i];
        pos = i;
    }
printf("The largest ele = %d", big);
printf("Its position is %d", pos);
}
```

Same program with pointers:

```
#include<stdio.h>
void main()
```

```
{
    int a[10], n, i, big, pos;
    printf("Enter n");
    scanf("%d", &n);
    printf("Enter the elements");
    for(i=0; i<n; i++)
        scanf("%d", a+i)
                  ↳ &a[i]
    big = *(a+0)
    pos = 0;
    for(i=1; i<n; i++)
    {
        if(big < *(a+i))
        {
            big = *(a+i);
            pos = i;
        }
    }
    printf("The largest %d", big);
    printf("Position = %d", pos);
}
```

Pointers and other operators:

Consider

```
x = *p1 * *p2;
```

✓

The value pointed by p1 * p2 or multiplied

```
sum = sum + *p2;
```

✓

The value pointed by p2 is added to sum.

```
*p1 = *p1 + 1;
```

✓

The value pointed by p1 is incremented by 1.

```
x = *p1 / *p2;
```

✗

/* → Beginning of the comment

```
x1 = *p1 / *p2;
```

✓

/ * X

Operations performed on Pointers:

- Adding an integer to a pointer ✓
- Subtracting an integer from pointer ✓
- Subtracting two Pointers ✓
- Comparing two Pointers. ✓

Adding an integer to a pointer:

Ex: int a[5] = {10, 20, 30, 40, 50};

int *p1, *p2;

```
p1 = a;
p2 = a;
```

p1 = p1+1; ✓ Points to the next element.

`p1 = p1 + 3;` ✓ points to 3rd element from p_1.

`p1 = p1 + 6;` ✗ Invalid

`a[5]`

`p1 + p2;` Invalid, Two pointers can't be added

`*p1 + *p2` ✓

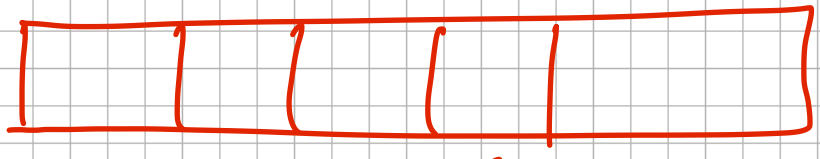
`p1++;`

void

`p1 = p1 + 1`

`p1++;` `p1 = p1 + 1` →

`p1--;` `p1 = p1 - 1` ←



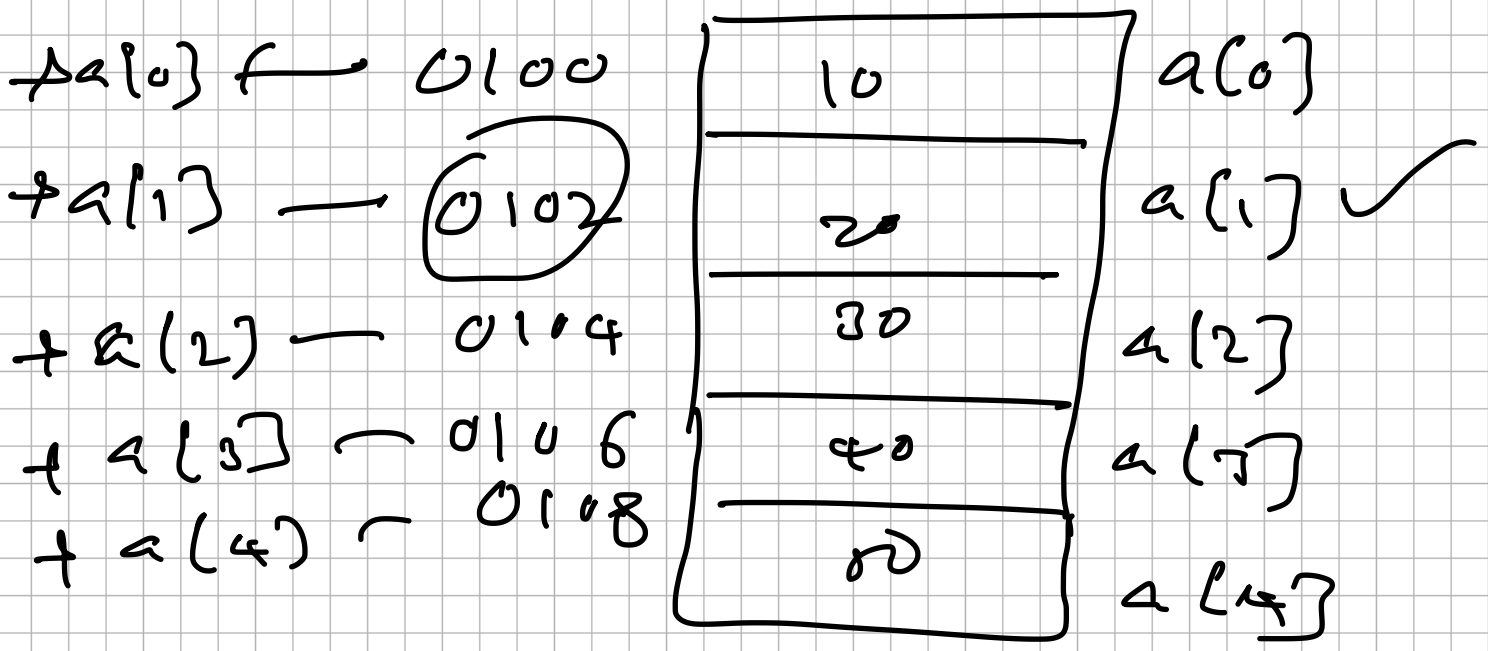
p_1 p_{1+1} p_{1+2}

→ `p1++` → `p1+1` `p1++`

Program to display array elements using pointers:

```
#include<stdio.h>
void main()
{
    int a[] = {10,20,30,40,50};
    int *p;
    int i;
    p = a;        [ p = &a[0] ]

    for(i=0; i<=4; i++)
    {
        printf("%d", *p);
        p++;
    }
}
```



P = &a[0]

Initially P = 0100

P++ = P = P+1 →

P = P+1

0101 X

0102 →

P = P+2

↑

↓

0106

0100
0102
0104
0106
0108

P = P+1 ⇒ Indicates the next memory location

0100+1

= 0101 ✗

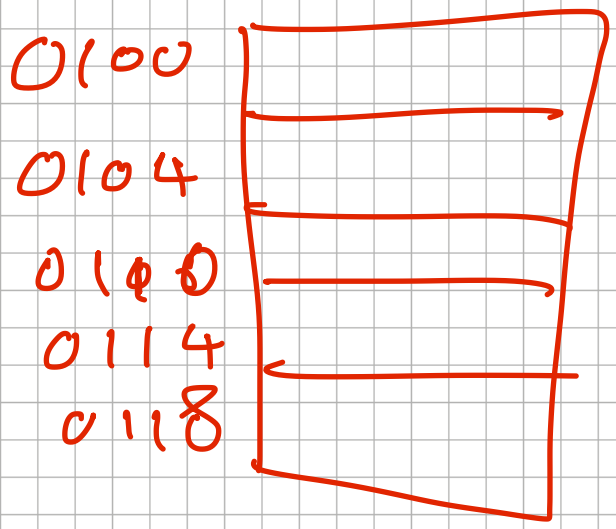
float a[7 = {10.0, 10.1}

p = a;

=

p = p+1

0104



Char → p = p+1 = 0100+1

= 0101

Int a

p = p+1 = 0100 → 0102

float

p = p+1 — 0100 → 0104

(K)

p = p+1 = 0100+K

Program to compute sum of elements of an array:

```
void main()
{
    int a[] = {10, 20, 30, 40, 50};
    int *p;
    int i, sum;
    p = a;          ( p = &a[0] )
    sum = 0;
    for( i = 0; i < 4; i++)
```

```
{
    sum = sum + *p;
    p++;
}
printf("The sum is %d", sum);
}
```

Note: observe that p++; means point p to the next element.

p++, p = p+1

Subtracting an integer from a pointer:

Subtracting can be performed when first operand is a pointer and the second operand is an integer.

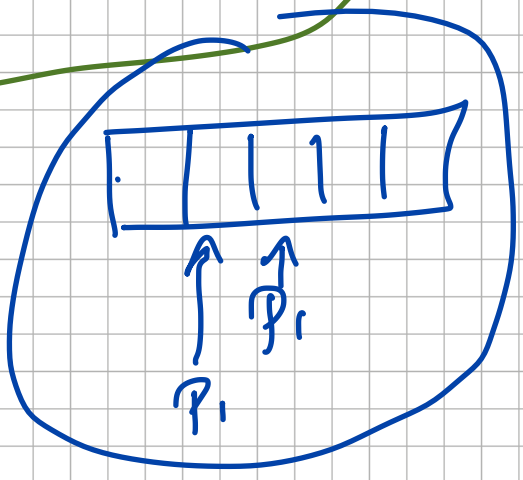
Ex:

Consider

```
int a[5] = {10, 20, 30, 40, 50};
int *p1;
p1 = &a[4];
```

```
p1 = p1 - 1;
```

valid




```
p1 = p1 - 3;
```

valid

p1--; valid ⇒ p1 = p1 - 1;

--p1; ⟶ valid p1 = p1 - 1

```
p1 = 1 - p1;
```

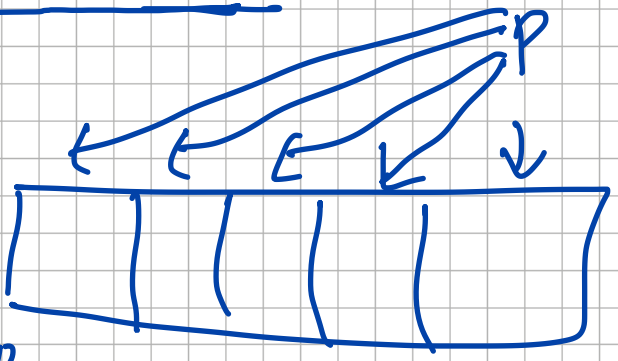
Invalid

*p1

Note: `p--` : pointer variable p points to the previous element.

Program to display array elements using pointer from last element to first...

```
void main()
{
    int a[] = {10,20,30,40,50};
    int *p, i;
    p = &a[4];
    for(i=0; i<=4; i++)
    {
        printf("%d", *p);
        p--;
    }
}
```



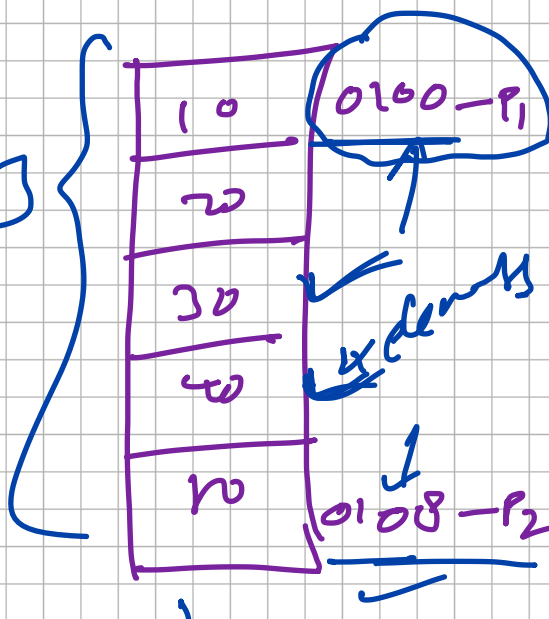
50, 40, 30, 20, 10

Subtracting two pointers:

If two pointers are associated with the same array, then subtraction of two pointers is allowed.

Consider the following declaration:

```
int a[] = { 10, 20, 30, 40, 50 };
int *p1, *p2;
float *f;
p1 = a;
p2 = &a[4];
```



p2 - p1; Valid

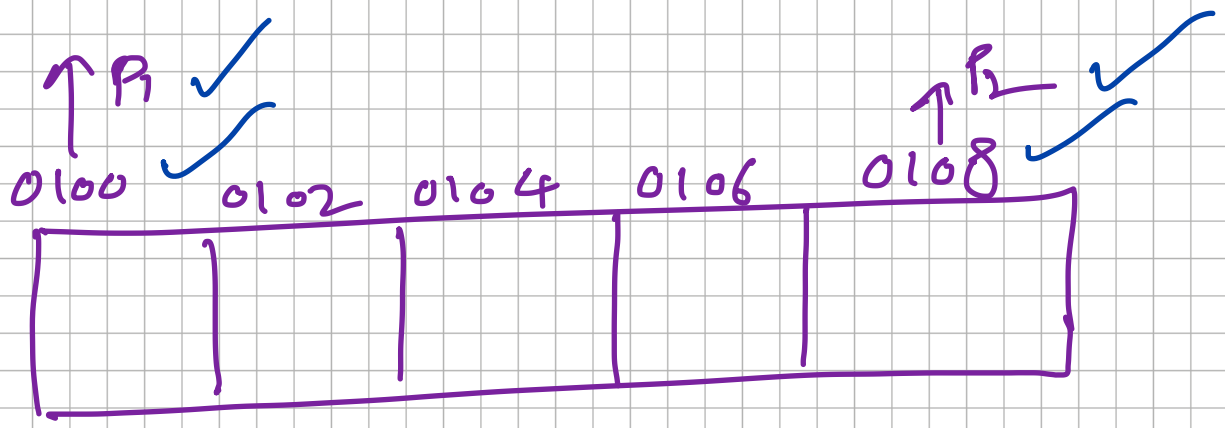
0108 - 0100 = 8

2 (4) 2

{ (p2 - p1)/2 + 1 }

p1 - p2; Valid

$f - p_1$; Invalid



from the above figure:

p_2 has an address — 0108

p_1 has an address — 0100

$p_2 - p_1$ → ?

$\frac{p_2 - p_1}{2} - 1$

$p_2 - p_1$

Hence, $p_2 - p_1$ gives us 4 which indicates that p_2 is at distance of 4 elements away from p_1

$\therefore \frac{p_2 - p_1}{2} + 1$ gives the no. of elements in the array

Comparing two Pointers:

If two pointers are associated with the same array, then comparison of two pointers is allowed using relational operators.

Consider example:

```
int a[5] = {10,20,30,40,50};
int *p1, *p2;
float *f;
p1 = a;
p2 = &a[4];
```

p1 → a[0]

Valid / Invalid ⟶ ?

```
p2 != p1;
```
valid ✓

```
p1 == p2;
```
valid

```
p1 <= p2;
```
valid

```
p1 >= p2;
```
valid

```
p != p1;
```
Invalid

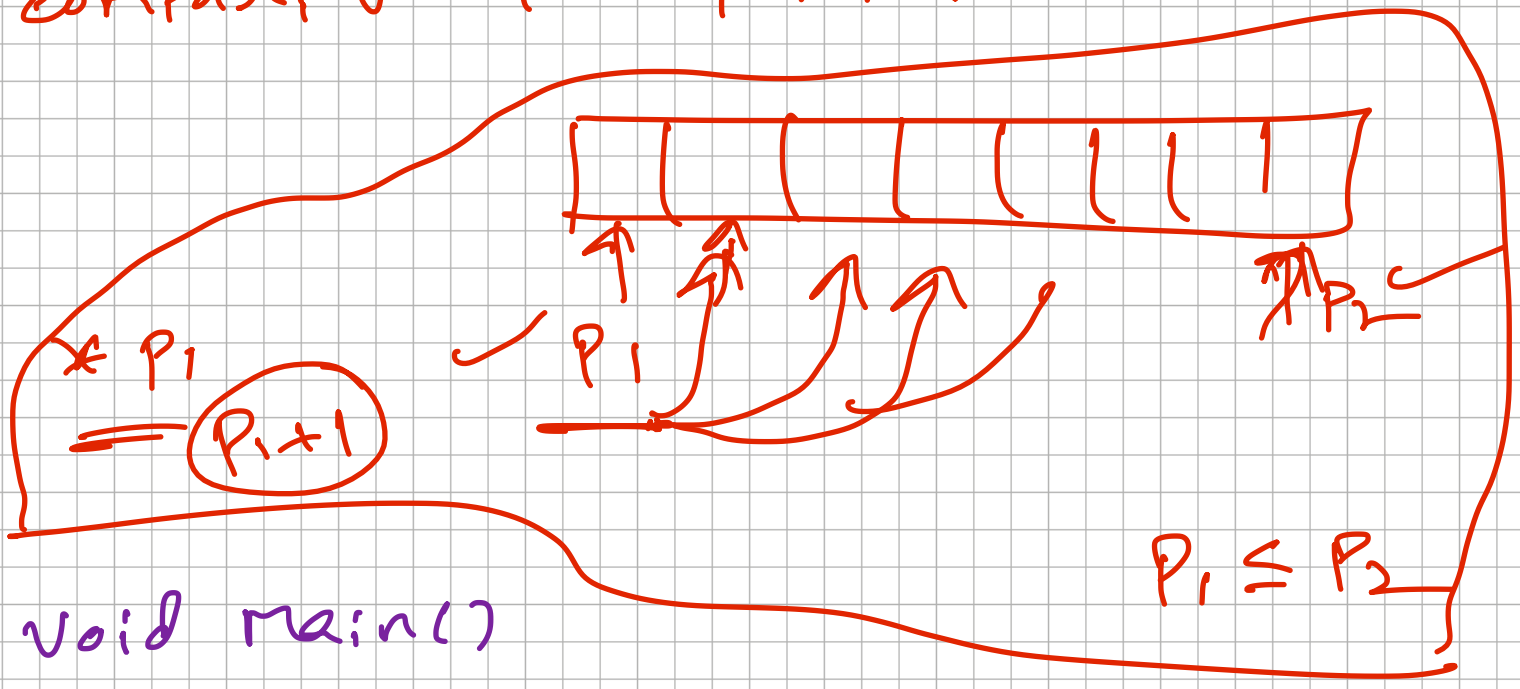
p1 = q

↓

q[0]

Note: Multiplying and dividing a pointer variable with any other variable or integer is not allowed.

Program to display array elements by comparing two pointers:




```
void main()
{
    int a[] = {10, 20, 30, 40, 50};
    int *p, *q;
    p = &a[0];
    q = &a[4];
    while (p <= q)
    {
        printf("%d", *p);
        p++;
    }
}
```

p1 = &a[0]

Passing an array to a function or character pointer and functions:

Name of an array is a pointer to the first element. So, when we pass an array to a function we should not use the address operator.

The syntax of a function call is:

```
function_name(a);
```

```
/* a is an array
   variable */
```

The two ways of declaring and using the array in the called function are:

- Using pointer declaration — [1]
- Using array declaration — [2]

[1]:

```
void function_name(int *a)
{
    /* ith element can be
       accessed using *(a+i) */
}
```

[2]:

```
void function_name(int a[])
{
    /* ith element can be accessed
       using a[i] */
}
```

Memory allocation functions:

Memory can be reserved for the variables either during compilation or during execution time (runtime).

Memory allocation techniques:

- Static allocation ✓
  (User declarations & definitions)
- Dynamic allocation ✓
  (using predefined functions).

Static memory allocation:

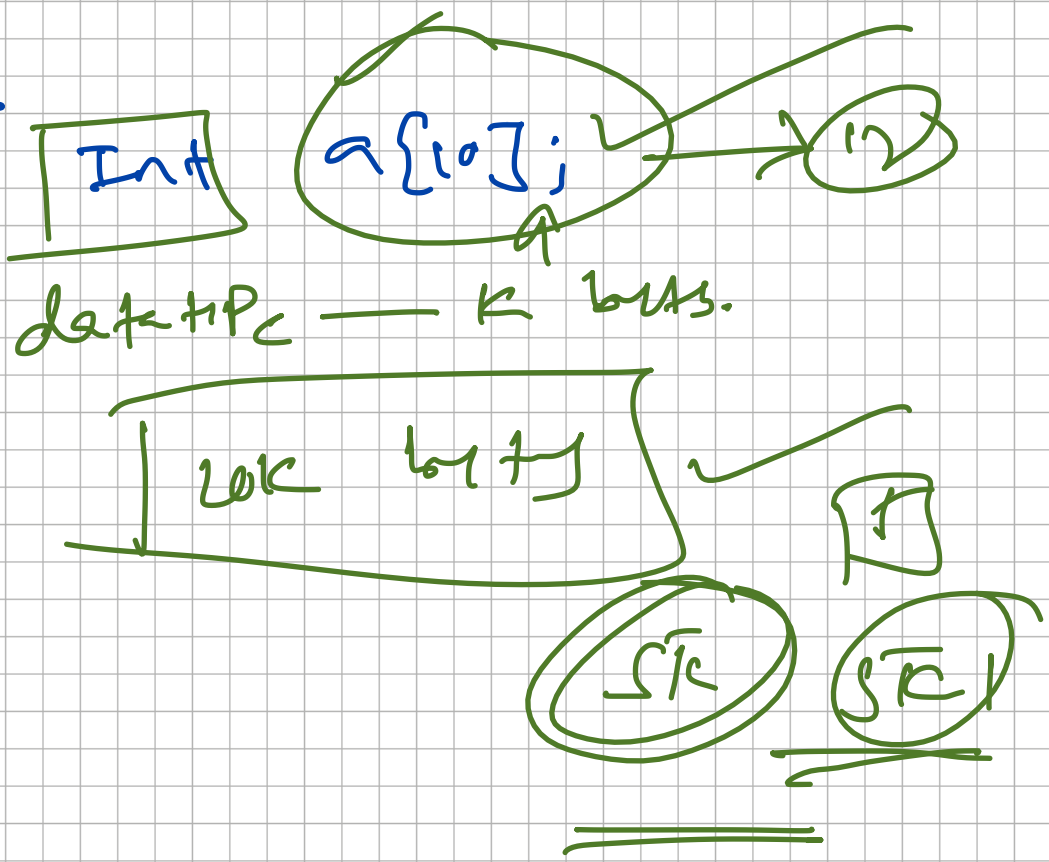
If the memory is allocated i.e, reserved for various variables during compilation time itself, the allocated memory space can't be expanded to

accommodate more data or can't be reduced to accommodate less data.

In this technique, once the size of the memory allocated is fixed, it can't be altered even during execution time.

This method of allocating the memory during compilation time is called "static memory allocation.

Ex:



Dynamic memory allocation:

It is the process of allocating memory during execution time (running).

NOTE: Unique feature of C when compared with other high level languages.

This allocation technique uses predefined functions to allocate and release memory for data during execution time.

If there is an unpredictable storage requirement, then the dynamic allocation technique is used.

Differences b/w Static & Dynamic memory allocation:

Static	Dynamic
1. compilation time	1. execution time
2. Can't be altered	2. altered.
3. Predictable	3. unpredictable
4. Faster	4. slower
5. memory is allocated either in stack (for local variables) or data area (for global + static variables) Ex: Arrays	5. memory is allocated only in heap area. Ex: LL, Array dynamic arrays.

Memory Management Functions in C:

- malloc()
  (Allocated a block of memory)
- calloc()
  (contiguous allocation of multiple blocks)
- realloc()
  (Reallocation of memory)
- free()
  (de-allocate the allocated block of memory)

---

malloc(size):

Syntax:

```
#include<stdio.h>
```

```
ptr = (datatype *) malloc(size);
```

NULL

```
void function_name()
{
    ....
    ptr = (datatype *) malloc(size);
    if( ptr == NULL )
    {
        printf("Insufficient memory");
        exit(0);
    }
    ...
}
```

A program to demonstrate malloc();

```
#include <stdio.h>
#include <stdlib.h>
void ram()
{
    int i, n, *ptr;
    printf("Enter the no. of elements");
    scanf("%d", &n);
    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL)
    {
        printf("Insufficient memory");
        return;
    }
    printf("Enter %d elements", n);
    for(i=0; i<n; i++)
        scanf("%d", ptr+i);
```

```
printf(" The elements are ");
for (i=0; i<n; i++)
  printf("%d", *(ptr+i));
}
```

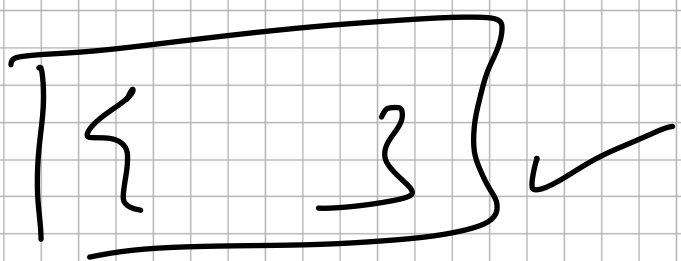
*ptr → a[i] → *(a+i)

calloc(n, size):

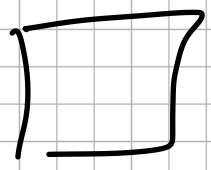
Syntax:

```
..... #include<stdlib.h>
.....
ptr = (datatype*)calloc(n,size);
.....
```

n → no. of units for each block.



Program to find maximum of n no.'s using dynamic array.



```
void main()
{
    int i,j,n,*a;
    printf("Enter the no.of elements");
    scanf("%d",&n);
    a=(int*)calloc(n,sizeof(int));
    if(a==NULL)
    {
        printf("Insufficient");
        return;
    }
    printf("Enter %d elements",n);
    for(i=0;i<n;i++)
```

scanf("%d", &a[i]);

11

a+i

j = 0;

for(i=1; i<n; i++)

if(a[j] < a[i])

j = i;

if(*(a+j) < *(a+i))

j = i;

printf("The biggest element is %d

at position %d", a[j], j+1);

free(a);

}

Explanation here