Date. No.

Pointers in C :

Pointers are special variables containing address of any other variables.
They are derived datatypes.

*) Declaration of Pointer:

Syntax: datatype * pointername ;

Eg: int * p; int p;

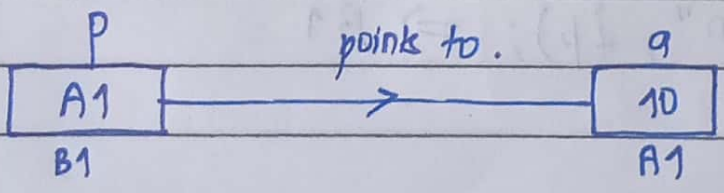
Here, p is pointer value Here, p contains integer containing address of integer value
type of variable.

*) Size of pointer: In 32 - bit, 4 bytes
In 16 - bit, 2 bytes.

*) Initialization of pointer:

pointername = & variablename.

Eg: int a, *ptr; float b;
a = 10;
*ptr = &a; *ptr = &b (wrong as ptr takes int addre



The address of variable in memory is in hexadecimal form.

In a single step,

```
int *p = &a
```

But

```
int *p=&a, a=10   is wrong:
```

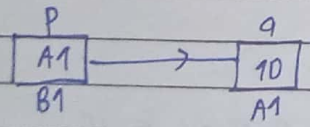
To declare a pointer, one has to first declare a variable.

(*) Operators used in Pointer operation:

The operators used in pointer operations are:

i) Address of (&) i.e, reference operator.
ii) Indirection operator (*)

Eg:
```
int a=10, b=9, *p=&a, *q=&b;
printf ("Value of a\n");
printf ("%d\n", a);   => 10
printf ("%d\n", *p);  => 10
printf ("Address of a\n");
printf ("%x\n", &a);  => A1
printf ("%x\n", p);   => A1
printf ("%x\n", &p);  => B1
```

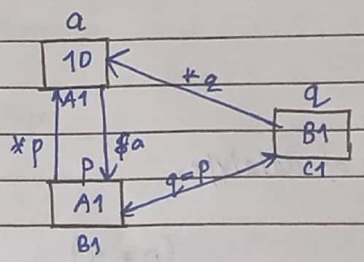


(*) Pointer Assignment

The process of assigning a pointer for a variable or for another pointer is called pointer assignment.

Eg:
```
int a = 10, b = 11
int (*)p; (*)q;      -> Here, * is not indirection operator. but resembles a pointer.
p = &a;
q = p;
printf("%d %t, %d %t, %d, %t", a, (*)p, (*)q);   -> Here, * is indirection operator as it points to the value at address stored.
```

Output: 10 10 10



(*) Pointer Arithmetic:

Pointer arithmetic is the method of calculating the address of an object with the help of arithmetic operations upon pointers.
It is also called address arithmetic.

Date. No.

(i): Pointer addition:
Operator used: +

We can add integer value to a pointer but we cannot add two pointers.

Eg: int a = 10;
int *p = &a;
int *q = &b;
p+q (x) → considered invalid in C.

However, p = p+n is valid.
formula: p = p + n * size of datatype.

Here, p = p+1 (n*4)
means increasing the address by 1x4 = 4 bytes.

Eg: int a[] = {0, 1, -1, 10, 11}
int *p = &a[0]

0	1	-1	10	11
1000	1004	1008	1012	1016

printf("%d", *p); ⇒ 0
p = p+2; // increase position by 2*4 = 8 bytes
// 1000+8 = 1008. ie, a[2]
printf("%d", *p) ⇒ -1

So, p = p+x ⇒ p = &a[0+x]

Date. No.

ii) Pointer Subtraction:
Operator used: -

We can subtract pointers or subtract integer value from pointer.
Subtracting pointers is only valid when its possible. but is not usually practiced.

Eg. int a[] = {0, 1, -1, 10, 11}
int *p = &a[0], *q = &a[3];
q = q-2;
Formula: p = p-n;
p = p - (n * size of datatype)
Here, = 1012 - (2*4) = 1012 - 8 = 1004
∴ q = 1004.
So,
printf("%d\n", *q); ⇒ 1

iii) Pointer Increment / Decrement:
Operators used: -- (decrement) ++ (increment)
This increases or decreases the address by 1.
Pointer increment and decrement are also of types: prefix and postfix.

Eg: int a[] = { 3, 2, 67, 0, 56 }
int *p = &a[0] // p = a;
p++;
printf("%d", *p) ⇒ 2
printf("%d", *p++) ⇒ 2 // post-increment.
printf("%d", *p) ⇒ 67
printf("%d", ++*p) ⇒ 0

(iv) Pointer comparison:

For pointer comparison, we use relational operators.

Eg:

(*) Passing pointer as an Argument to Function:

The method of passing pointer as an argument to function is called calling function by reference.

Eg: Syntax: Declaration:

datatype functionname (datatype*, datatype*);

Definition:

datatype functionnaem (datatype* variablename, datatype* vname)
{ ..... }

calling:

functionname (address of variable, address of variable);

Eg:

```
int add (int*, int*);
void main()
{
  int a, b, sum;
  a=10, b=5; sum = add(&a, &b);
  printf("Sum = %d", sum);
}
```

```
int add (int *x, int *y)
{ int z;
  z = *x + *y;
  return z;
}
```

Output: Sum = 15

(*) Passing Array pointer as an Argument to Function

Declaration: datatype functionname (int*, int);

Definition: datatype functionname (int* arr, int size)
{ ..... }

Calling: functionname (arrayname, length);

(*) Void pointer:

The pointer that has no associated datatype and whose datatype can be changed is called void pointer.

→ Since void pointer is a generic pointer, it can be converted to any other type of pointer through typecasting.

Syntax: void * pointername;

Eg: void *vp;
int * a=5, *ip;
vp = &a;
char *c='0', *cp;
float *fp;

printf ("%d", *vp); ⇒ gives Error.

⇒ Void pointers cannot be dereferenced. Hence, we need to use typecasting.

printf ("%d", *(int*) vp); ⇒ 5
vp = &c;
printf ("%c", *(char*) vp); ⇒ 0

Dynamic Memory Allocation:

The process of allocating the memory at run time is called dynamic memory allocation.

*) Advantages:

i) Efficient memory usage
ii) allocation and deallocation of memory can be done as pleased.

*) Disadvantages

i) Takes more time
ii) Memory must be free after by the users.

*) Static Memory Allocation:

The process of allocating memory during compile time is called static memory allocation.

*) Differences: DMA and SMA:

Static Memory allocation	Dynamic memory allocation.
Memory allocated at compile time.	– Memory allocated at run time.
Memory can't be changed while executing program.	– Memory can be changed while executing program.
It is quicker than DMA.	– It is slower than SMA.
It allots memory from stack.	– It allots memory from heap.
It is less efficient.	– It is more efficient.
Memory allocation is simple.	– Memory allocation is complex
Allotted memory remains from beginning to end of program.	– Memory allotted and freeing can be done at any time.

DMA uses <stdlib.h> header file.

Date. No.

(*) Memory layout:

Memory layout has four segments:

Heap	⇒ big pool of free memory
stack	⇒ for local variables and functions
Global	⇒ for global variables and functions
Code	⇒ for instructions.

Dynamic memory allocation is done by using memory provided in heap section.

The functions used during DMA are as follows:
i) malloc ii) calloc
iii) realloc iv) free.

i) malloc:

Fullform: memory allocation.
It is generally used to allocate memory to structures.

Malloc allocates a block of memory from heap section and returns the base address of the block as pointer of type void.

Syntax: (type *) malloc (byte_size)
typecasting returns void pointer.

Eg: int *p;
p = (int *) malloc (100 * sizeof (int));

Here, p contains the base address of the assigned memory block.
and
malloc assigns 400 byte of contiguous memory as a single block.

Eg: For structure,
int *sptr;
sptr = (struct stud *) malloc (n* sizeof (struct stud));

If memory values is not provided, it gets initiated with garbage value.

If malloc can't assign a memory block ie, on failure, it returns a null pointer.

ii): Calloc:

fullform: contiguous allocation.
It is generally used to allocate memory to structures.

It is used to dynamically allocated multiple block of memory and each block is of same size.
It's return type is void pointer.

Syntax: (type *) calloc (no.of blocks, sizeof a block);

Eg: int *ptr;
ptr = (*int*) calloc (5, sizeof(int));

Here,
ptr returns the base address of the block.

If the memory is not initialized, every block has value 0.

On failure, calloc returns a null pointer.

(*) Difference: malloc and calloc

malloc()	calloc()
- It creates a single block of memory of specific size.	- It assigns multiple blocks to single variable.
- fullform: memory allocation	- fullform: contiguous ~~mem~~ allocation.
- malloc is faster	- calloc is slower.
- malloc allocated block has garbage value.	- calloc allocated block has 0 initialized.
It takes ~~two~~ one arguments	- It takes two arguments.
Syntax malloc (size);	- Syntax: ~~calloc~~ calloc (no.of block, sizeof a block);

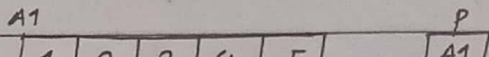
(iii) realloc:

Fullform: re allocation.

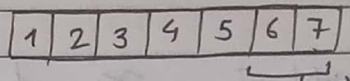
It alters the size of previously allocated block without losing the previous content.

(type3*)
Syntax: realloc (name of previous pointer, new size);

Eg: int *p, *rp;
p = (int*) ~~re~~alloc (5, sizeof(int));



rp = (int*) realloc ~~(7 * size)~~ (p, 7*sizeof(int));



New no.of blocks = 7 ie, increased by 2.

Realloc expands the same block of memory if possible.
If it is not possible, it allocates a new block, ~~alloca~~ copies the previous content, frees the old block and returns the base address of the new block.

Date. No.

If reallocation of memory in heap is not possible, it returns NULL pointer and original block is freed.

(iv) free:

It is used to deallocate the memory allocated by malloc, calloc and realloc.

Syntax: free (pointername);

*) Note:

i) height +i ≈ & height [i]

ii) *(height +i) ≈ height [i]