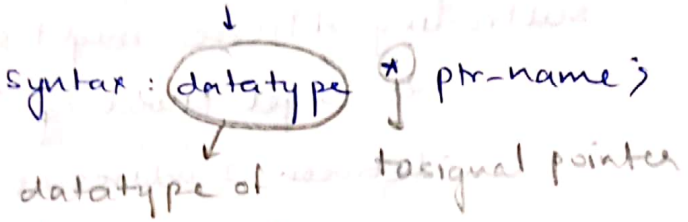
Advanced * C programming language.

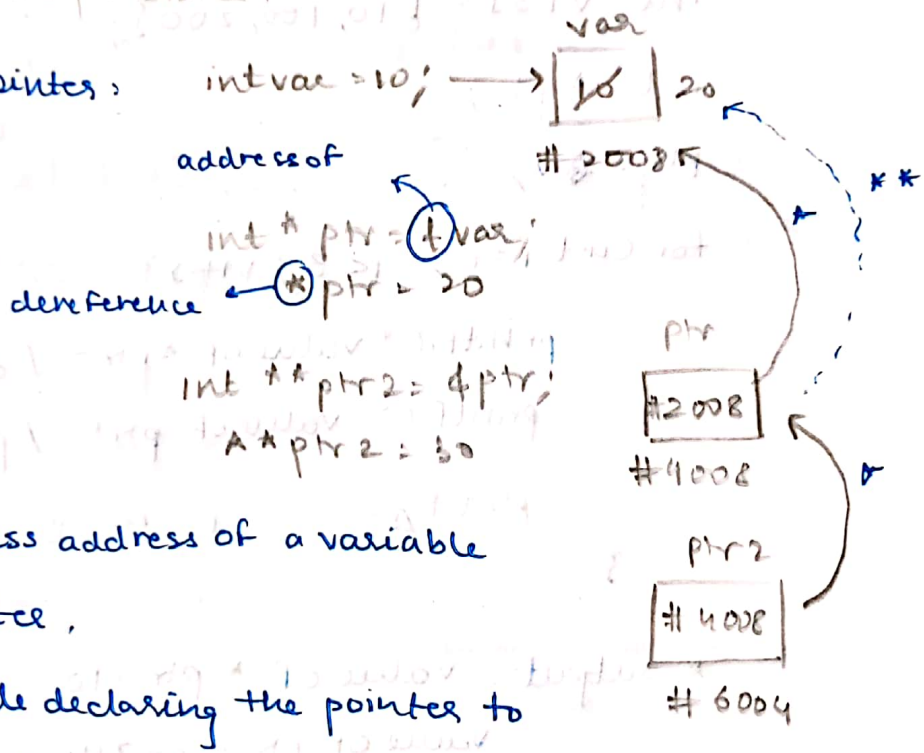
Pointers:

1) Introduction, Arithmetic and Array.

*) pointers store address of variables or a memory location



*) using a pointer:



& : to access address of a variable to a pointer,

* : i) While declaring the pointer to indicate that this variable will store an address.

ii) to access value stored at the address stored in the ptr or pointed to by the pointer.

*) pointer expressions and pointer arithmetic

i) ++
ii) --
iii) +, or +=
iv) -, or -=

i) pointer arithmetic makes sense for arrays

ii) pointers contain addresses, their addition does not make sense, because there is no logic what they would point to.

↓

subtracting addresses might allow the user to get knowledge of offset between 2 addresses

```
int v[3] = {10, 100, 200};
int *ptr;
ptr = &v;
for(int i=0; i<3; i++){
    printf("value of *ptr = %d\n", *ptr);
    printf("value of ptr = %p\n", ptr);
    ptr++;
}
```

* output: value of *ptr = 10
value of ptr = 0x7ffcae30c710

value of *ptr = 100
value of ptr = 0x7ffcae30c714

value of *ptr = 200
value of ptr = 0x7ffcae30c718

*) Array name as pointers

↓

acts as a pointer constant

↓

address of the first element

↓

i.e val & val[0] can be used interchangeably.

more efficient during passing to arrays.

eliminates the need to copy data of array & the same array can be referenced.

*) Pointers and multidimensional arrays

int nums [2][3] = {{16, 18, 20}, {25, 26, 27}};

(nums)

↓

nums [0][0]

((nums+1))

↓

nums [1][0]

*(*nums+1)

↓

nums [0][1]

*(*nums+2)

↓

nums [0][2]

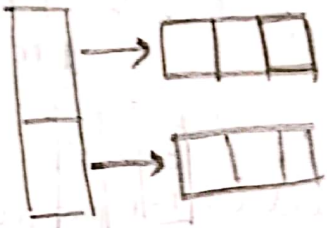
((nums+1)+2)

↓

nums [1][2]

(array of ptrs to other arrays)

nums.



nums.

16	18	20	25	26	27

nums stores the address of the 0th index of the array

2) Double pointer (pointer to pointer) in C

pointer storing address of another pointer

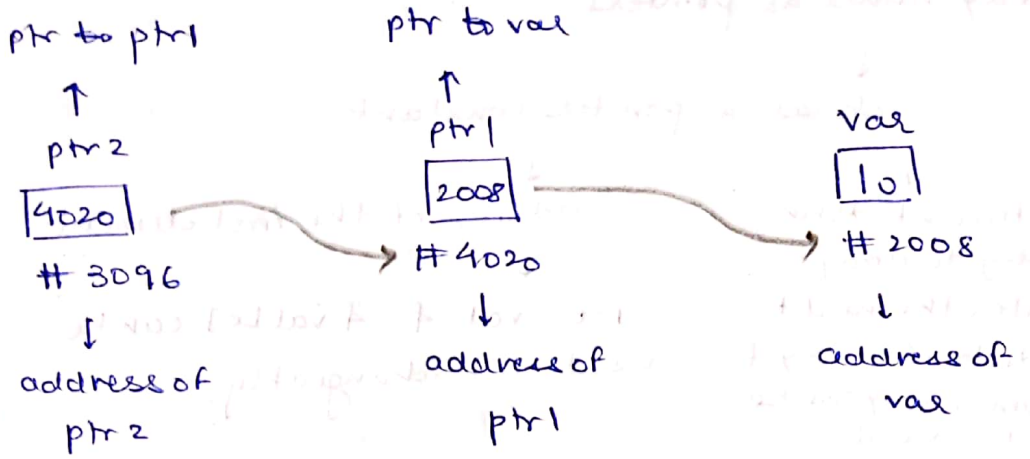
int * (*) ptr → pointer name

pointer indicator

↓

type of data

the address of which the ptr stores




```
int var = 10
int *ptr1;
    ptr1 = &var;
int **ptr2;
    ptr2 = &ptr1;
```

4000

3) predict o/p of program

ptr # e001

e000

```
#include<stdio.h>
int main(){
    char *ptr = "geeks for geeks";
    printf("%c\n", **&*ptr);
}
```

*ptr = g

&*ptr = &g (# 4000)

*&*ptr = g

&*&*ptr = &g

&&*ptr = g

%p → format for pointer → hexcode

printf("%c\n", **&*&ptr);

(address of pointer ptr)

(due to %c, &)

(implicit type conversion)

ptr → # 8000

&ptr → # 4000

*&ptr → # e000

&*&ptr → # 4000

&&ptr → # e000

4) Dangling, void, null & wild pointers.

pointers pointing to a memory location that has been freed is called dangling pointer.

↓

i) De-allocation of memory

```
int * ptr = (int *) malloc (sizeof (int));
free (ptr)
ptr = NULL
```

dangling pointer here

↳ no more a dangling pointer

↳ malloc returns void pointer

↓

needs to be explicitly typecasted

ii) Function call

↓

pointers pointing to local variable becomes dangling, unless it is static, since when function finishes execution, local function memory is freed. Next time it is called, new memory, possible other than the one previously assigned will be put.

variable goes out of scope

```
int * fun() {
    int x = 5;      → static int x = 5;
    return &x;
}
```

has scope throughout the program

* void pointer: pointers pointing to some data location which doesn't have a specific type.

↓

The type of data it refers to can be any.

↓

if address of char is assigned, then points to char,
if address of int is assigned, then points to int.

any pointer is convertible to void pointer

↓

void pointers can't be dereferenced straightaway. Need to be typecasted to some type first.

↓

pointer arithmetic is not applicable on void pointers. (okay for some compilers with size = 1)

↓

malloc & calloc use void pointers to be able to allocate memory of any data type.

```
int * x = malloc (sizeof(int) * n)
```

↓

compiles in C, not in C++

↓

needs to be explicitly typecasted.

↓

used to implement generic functions in C, as well (compare function in qsort())

↓

this as well. because it needs to be applied on different kinds of arrays.

5) compiler converts any operation to pointers before dereferencing. i.e.

arr[0] & 0[arr] are same

and are converted to (arr+0)* or (0+arr)*

1 [arr+1] → * (arr+2);

6) pointer to a function

pointer declaration : datatype * name;

function declaration : int foo(int);

↓

binding ^ to function name

(Y.d → int)

int (*foo)(int); and not

int *foo(int)

↓

returns a int pointer, and does not actually bind ptr to fn

for example:

```
void fun(int a) {
    print(a);
}
int main() {
    void (*fun_ptr)(int) = &fun;
    (*fun_ptr)(10);
}
```

(invoking function using function pointer)

↓ important points

i) unlike normal pointers, a fn. pointer points to code and not to data. Typically, a function ptr stores the start of the executable code.

ii) deallocation of memory is not done using fn. pointers

iii)

```
void (*fun_ptr)(int) = fun;
fun_ptr(10);
```

} also works.

iv) array of fn. pointers can also be made.

```
void add(int a, int b) print(a+b);
void subtract(int a, int b) print(a-b);
void multiply(int a, int b) print(a*b);
int main(){
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    a = 2;
    b = 3;
    fun_ptr_arr[0](a, b);
}
```

important to fn pointers

pointer to array of functions.

since function names also act as address of functions.

v) can be passed as argument and can also be returned by a function.

```
void fun1() { print("fun1"); }
void fun2() { print("fun2"); }
void wrapper(void (*fun)()){
    fun();
}
int main(){
    wrapper(fun1);
    wrapper(fun2);
}
```

used by function like qsort to sort elements in ascending or descending order, or can be applied on different structs, even own functions to customize sorting criteria can be defined.

vi) Virtual functions of class methods in c++ implemented using function pointers.

7) Pointer vs Array in C

↓

Most of the time, pointer & array accesses can be treated same, except for the following cases:

i) sizeof operator: sizeof(array) returns amount of memory allocated to entire array

sizeof(ptr) returns amount of memory allocated for any pointer.

ii) & operator: &array is an alias for &array[0] returns address of 0th element (same as arr)

&ptr: address of ptr.

(**)

iii) string literal initialization of character array

```
char array[] = "abc";   (first 4 elements. a, b, c, \0)
char *ptr = "abc";
```

↳ sets the ptr to address of "abc" string.

iv) ptr variable is changeable unless constant. not the case in array.

```
int a[10];
int *p;
p = a;   → legal
a = p;   → illegal
```

(array passed as pointer to functions. ↓ sizeof returns 8 in functions.)

(increment)

v) pointer arithmetic allowed on ptr and not on array pointer.

```
p++;   ✓
a++;   ✗
```

8) Null ptr: An integer constant expression with the value 0, or such an expression cast to type void* is called a null pointer constant.

↓

when converted to a pointer, will return unequal to any pointer of any type.

↓

(pointer equivalent of 0)

↓

(does not reference to any memory)

↓

sizeof (NULL) dependent on platform (memory allotted to pointers)

↓ major uses:

a) to initialise a ptr variable.

b) to be able to check conditionally for non-erroneous dereferencing.

c) passing to a fn.

9) const char *p , char * const p and const char * const p.

const applies to whatever is on its left, if nothing present on left, then whatever is present immediately to the right.

1) const char* p : point to constant char

↓

value pointed by the pointer cannot be changed, but the value of the pointer can be.

const char *p

char const *p

↓

same

ii) char * const p : ptr is constant ∴ ptr cannot be reassigned. the value at the address the pointer points to be can be changed though.

iii) const char * const ptr : const ptr to const variable

10) pointer to an array (treats array as a normal datatype)

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

↓

Here pointer points to the 0th element of the array.

↓

but pointer to complete array can also be defined. instead the 0th element only.

↓

useful when dealing with multidimensional arrays.

↓

syntax : data type (*var-name) [sizeof array]

```
int (*ptr)[10];
```

ptr is a pointer to array of 10 integers

↓

```
int *p;
int (*ptr)[5];
int arr[5];
p = arr;
ptr = &arr;
```

p++;
ptr++;

p = ... 00 → p = ... 04 (moves by 4)

ptr = ... 00 → ptr = ... 14 (moves by 20 (pointer arithmetic))

hex code 16+4 = 20

pointer arithmetic performed relative to the base size. :- ptr++ resulted in shift by 20 bytes.

↓

*ptr → address of the array's 0th element is provided.

↓

on dereferencing a pointer, we get a value pointed to by that pointer expression.

↓

pointer to an array points to an array, so on dereferencing it, we should get the array.

↓

name of array denotes the base address.

↓

```
int arr[5] = {3, 5, 6, 7, 9};
int *p = arr;
int (*ptr)[5] = &arr;
print (p, ptr);          ...5010      ...5010
print (*p, *ptr);        3            0x7ffdeleec5010
print (sizeof(p), sizeof(ptr));   8, 8
print (sizeof(*p), sizeof(*ptr));   4, 20
```

* pointers to multidimensional arrays

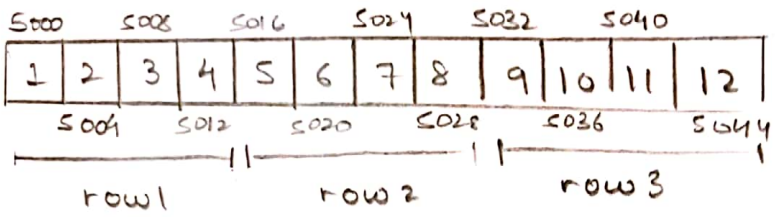
↓

accessing elements using pointer notation.

arr[i][j] = *(*(arr+i)+j))

↓

. memory in computer organized linearly. 2D arrays stored in row-major order.



↓ collection of 1D arrays.

↓ arr consists of 3 elements where each element is an array of integers.

int (*ptr)[5] = arr;

ptr[i][j] also works.

(subscripting pointer to an array)

* arr stored 5000, Now arr+1, results in size of the data at element, which is 16 here, ∴ arr+1 = 5016, arr+2 = 5032

↓

Now dereferencing these return these addresses only but now to the 0th element.

∴ *((*arr+1)+2)

↓

arr[1][2];

Since arr+i points to the ith element of arr, dereferencing it will get ith element of arr, which is 1D array.

* in case of 3D array: array of 2D arrays. Now each element of 3D array refers to complete 2D array.

(((*arr+i)+j)+k) → reference an element of the array.

↓ reference 2D array; ↘ reference 1D array