POINTERS (Lec 16 - Lec 18)

Pointer Initialization

ptr_var = &var;

Eg:
```
int x = 10;
int *ptr = &x;
printf("%d", *ptr);   → 10
printf("%p", ptr);    → 0x16da0b1cc
```

int* → is a type

Obtaining value stored at that location is known as dereferencing the pointer.

& - "Address of operator" OR "Referencing Operator"
(Taking address of an existing variable to set a ptr var)

* - "Dereference Operator"
(To get the value from the memory address that is pointed by the ptr.)

* - Value ~~of~~ at address operator OR indirection operator.

Eg:
```
int i = 3;
printf("%u", &i);      → 65524
  "(%d, i);            → 3
  "(%d, *(&i));        → 3
```

The value of variable which is pointed by a ptr can be accessed & manipulated using ptr variable.

```
int x=10;
int * ptr = &x;
printf("%d\n", *ptr);   → 10
*ptr = 20;
printf("%d\n", *ptr);   → 20
```

Size of a Pointer

→ Size of ptr in C is equal for every ptr type.

→ Size of ptr DOES NOT depend on type it is pointing to.

→ Size of ptr :- 8 bytes for 64-bit system
4 bytes for 32-bit system

int* → type

Eg
```
int var = 10;
int* ptr;
ptr = &var;
```
} `int * ptr = &var;`

```
printf("%p\n", ptr);   → 0x...
printf("%d\n", var);   → 10
printf("%d\n", *ptr);  → 10
```

Print value and Address of an integer

```
int var = 10;
int * p
printf("Address : %p", &var);   → 0x...
printf("Value : %d", *var);     → 10
```

Null ptr

Eg.
```
int *ptr = NULL;
printf("%x\n", ptr);  → 0
printf("%p\n", ptr);  → 0x0
```

Ptr to Ptr

```
int a = 10;
int *x = &a;
int **k = &x;
```

	a	x	k
	10	65524	65522
	65524	65522	65520

&a = x = *k

&x = k

a = *(&a) = *x = **k

Pointer Arithmetic

A ptr var stores address of another variable

The address is always an integer. So, we can perform arithmetic operations.

→ Increment & Decement of a Ptr (++ & --)

Eg.
```
int x = 10;
int *y = &x;
printf("%d\n", y);  → 1807315404  → 0x
y++;
printf("%d\n", y);  → ...408
y--;
printf("%d\n", y);  → ...404
```

} Diff. of 4 as x is an int
size of (int) = 4

→ Addition & Subtraction of Integers to a Pointer

Eg.

```
int arr[] = {12,23,45,67,89};

int *ptr = &arr[3];

printf("%d\n", *ptr);      → 67

ptr = ptr+1;
printf("%d\n", *ptr)       → 89

ptr = ptr-2;
printf("%d\n", *ptr)       → 45
```

→ Subtraction of Pointers

(When subtracting 2 ptrs, the result is the no. of elements b/w them)

Eg. int a[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int *x = &a[0];
int *y = &a[9];

printf("%ld %ld", x, y); x → 14.... 4896
y → 14.... 4932

printf("%ld", y-x); → 9 $\left(\frac{932-896}{4} = \frac{36}{4} = 9\right)$

printf("%ld", y-x+5); → 14 (9+5)

→ Comparison of pointer

```
const int MAX = 3;
int var[] = {10, 100, 200};
int i, *ptr1, *ptr2;
ptr1 = var;
ptr2 = &var[MAX-1]
while (ptr1 <= ptr2) {
  printf("Add of var[%d] = %p\n", i, ptr1);
  printf("Value of var[%d] = %d\n", i, *ptr1);
  ptr1++;
  i++;
}
```

OUTPUT:

Add of var[0] = 0x..
Value of var[0] = 10

Add of var[1] = 0x....
Value of var[1] = 100

Add of var[2] = 0x...
Value of var[2] = 200

Pointer to an Array OR Array Pointer

```
int arr[5] = {1,2,3,4,5};
int* ptr = arr;          → (points to 0th element of the array)

printf("%d", *ptr);      → 1
```

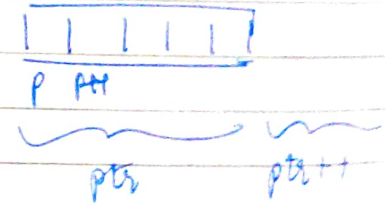
* We can also declare a ptr that can point to whole array

```
datatype (*varname)[sizeofarr];
```

varname → name of ptr variable

datatype → type of data array holds

Eg.
```
int (*ptr)[5];   → (ptr to an array of 5 integers)
int *p;          → (ptr to an integer)
int arr[5];
p = arr;         → (points to 0th element of the array)
ptr = &arr;      → (points to the whole array)
printf("p = %p, ptr = %p\n", p, ptr);
p++;                          1000   1000
ptr++;
printf("p = %p, ptr = %p\n", p, ptr);
return 0;                     1004   1020
}
```



Sizes of ptr of array

```
int arr[] = {3,5,6,7,9};
int *p = arr;
int (*ptr)[5] = &arr;
```

Size of ptr in C is equal for every ptr type

8 ↑ 4 (int) ↑

```
printf("sizeof(p) = %lu, sizeof(*p) = %lu\n", sizeof(p), sizeof(*p));
printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n", sizeof(ptr), sizeof(*ptr));
return 0;
```

sizeof(ptr) → 8

sizeof(*ptr) → 20 (5×4)

20 ↓ no. of elements, 4 ↓ int

Traversing an Array Using ptr

1D:

```
int a[] = {10, 20, 30, 40, 50};
int len = sizeof(a) / sizeof(int);
int *x = a;
int i = 0;

for(i=0; i<len; i++) {
printf("%d\n", *x);
x++;
}
```

OUTPUT:

```
10
20
30
40
50
```

2D:

```
int s[4][2] = {{1,2},{3,4},{5,6},{7,8}};
for(int i=0; i<=3; i++){
printf("\n");
for(int j=0; j<=1; j++) {
printf("%d", *(*(s+i)+j));
}
}
```

s[i] = *(s+i)

2D

Array of Pointers or Pointer Array

A ptr array is a homogenous collection of indexed pointer variables that are references to a memory location.

Eg
```
int var1 = 10;
var 2 = 20;
Var 3 = 30;

int* ptr_arr[3] = {&var1, &var2, &var3};

for(int i=0; i<3; i++){
 printf("%d", *ptr_arr[i]);
}
```

OUTPUT:
10 20 30

~~Pointer to access 2-D array elements~~

FUNCTION POINTERS (dec 19)

A var that stores the add of a fun is called fun ptr or a ptr to a function.

fun_return_type (*ptrname)(fun arg. list)

1 Eg.
```
void hello(){
  printf("Hello World");
}

void (*ptr)() = &hello;   → Declaring f^n ptr

(*ptr)();   → Calling f^n using ptr
```

2 Eg.
```
int add(int a, int b){
  return a+b; }

int main(){
int (*ptr)(int, int) = addition;   ← works w/o '&' also
int x=10, y=20;

int z=(*ptr)(x,y);   → can also be written as ptr(x,y) (* removed)
printf("%d", z);

}
```

→ Unlike normal code, f^n ptr points to code, not data. Typically a f^n ptr stores the start of the executable code.

→ We do NOT allocate, deallocate memory using f^n ptrs.

→ A f^n 's name can also be used to get function's address.
Removing '&' in assignment and '*' in f^n call, the program still works.

→ We can also hv arr of f^n ptrs

F^n ptr can be used in place of switch case.

Eg. void add(int a, int b){ . . . . .

(fnptr – switchcase.c)

→ Like normal data ptrs, a f^n ptr can be passed as an argument and can also be returned from a function.

Eg.
```
void fun1(){ printf("fun1\n"); }
void fun2(){ printf("fun2\n"); }

void wrapper(void (*fun)()){ fun(); }

int main(){

wrapper(fun1);
wrapper(fun2);

}
```

OUTPUT: fun1
fun2

qsort()

It is used sort arrays in asc. order or des. order or any other order in case of array of structures.

We can use qsort() for any data type.

(qsort.c)

Pointer to Function with Pointer Arguments

(lect 19_1.c)

Array of Function Pointers

(lect 19-2.c)

Void Pointer

→ A void ptr has no data type associated with it.

→ A void ptr can hold an address of any type and can be typecasted to any type.

```
void * ptr_name;
```

→ Void ptrs cannot be dereferenced.

Eg.

```
int a = 10;
char b = 'x';

void *p = &a;
printf("%d", *p);

p = &b;
printf("%c", *p);
```

ERROR: Invalid use of void expression

```
int a = 10;
char b = 'x';

void *p = &a;
printf("%d", *(int*)p);

p = &b;
printf("%c", *(char*)p);
```

OUTPUT: 10
x

Null Pointer

Pointer that does not point to any location but NULL.

```
type *ptrname = NULL;
```

Eg: `int * ptr = NULL;`

NULL POINTER	VOID POINTER
A NULL pointer does NOT point to anything. It is a special reserved value for pointers.	A void pointer points to the memory location that may contain typeless data.
Any ptr type can be assigned NULL.	It can only be of type void.
All NULL pointers are equal.	Void ptrs can be different
NULL ptr is a value.	Void ptr is a type.

Dangling Pointers

Ptrs that refer to a memory location that has been freed.

1. De-allocation of Memory

Dynamic Memory Allocation

Defined as the procedure in which the size of a data str. (like array) is changed during the runtime.

1. malloc() - Used to dynamically allocate a single large block of memory with the specified size.

It returns a ptr of type void which can be cast into a ptr of any form.

```
ptr = (int*) malloc (100 * sizeof(int));
      cast-type      size
```

the size of int is 4 bytes, this statement will allocate 40 bytes of memory.
The pointer ptr holds the address of the first byte in the allocated memory.

(malloc.c)

2. calloc()

Used to allocate specified number of blocks of memory of specified type.

Different from malloc() as:-

→ It initialises each block with a default value 0.
→ It has 2 parameters/arguments.

ptr = (cast-type*) calloc(n, element_size);

Here, n → no. of elements
element_size → size of each element

Eg. ptr = (float*) calloc(25, sizeof(float));

↳ Allocates contiguous space in memory for 25 elements each with the size of the float.

(calloc.c)

3. free()

It is used to de-allocate the memory.

The memory allocated using malloc() & calloc() is NOT deallocated on its own.

Syntax :- free(ptr)

4. realloc()

It is used to change the memory allocation of a previously allocated memory.

If the memory previously allocated is insufficient, realloc is used.

Re-allocation of memory maintains the already present value & new blocks will be initialized with the default garbage value.

ptr = realloc(ptr, newSize);

ptr is reallocated with new size 'newSize'.

int* ptr = (int*) malloc(5* sizeof(int));

20 bytes of memory

ptr = realloc(ptr, 10* sizeof(int));

40 bytes of memory

size of ptr is changed from 20 bytes to 40 bytes.

If space is insufficient, allocation fails & returns a NULL ptr.

(realloc.c)

Malloc()	calloc()
faster than calloc	slower than malloc
takes 1 argument	takes 2 arguments
used to indicate memory allocation	used to indicate contiguous memory allocation.
Does NOT initialise memory to 0.	Initializes memory to 0.
Does NOT add extra memory overhead	Adds some extra memory overhead.

STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
Memory is allocated at compile time	Memory is allocated at run time.
Memory can't be inc. while executing the program.	Memory can be inc. while executing the program.
Used in array.	Used in linked list.

Dangling Pointer

A ptr pointing to a memory location that has been deleted (or freed) is called a dangling pointer.

1. De-allocation of memory

When memory pointed by a ptr is deallocated the ptr becomes a dangling ptr.

(dealloc.c)

2. Function Call

When the local variable is NOT static and the function returns a ptr to that local variable.

The ptr pointing to the local variable becomes dangling pointer.

3. Variable goes out of scope

When var goes out of scope, the ptr pointing to that var becomes a dangling ptr.

eg. int main() {

int* ptr;

{ int a=10; // creating a block
ptr = &a;
}

printf("%d", *ptr); // ptr here becomes dangling ptr

}

OUTPUT: 10

Wild Pointer

A ptr that has NOT been initialised to anything is known as a wild ptr.

The ptr may be initialized to a non-NULL garbage value that may not be a valid address.

```
int main() {
int *p;
int x=10;
printf("%d\n", *p);
}
```

OUTPUT: Segmentation fault

1. Uninitialized Pointers : Using ptrs w/o assigning a valid address.

```
int main(){
int *p;
*p=10;
printf("%d\n",*p)}
```

OUTPUT: Segmentation fault

2. Correct way

```
int main(){
int x=10;
int *p=&x;

*p=20;

printf("%d\n", x);
printf("%d\n", *p);
```

OUTPUT: 20
20