Data Structures & Algorithms

28/9/2021

Structures :-

```
struct number
{   int a;
    char name[20];
};
struct number first, second;
```

first.a

second.a

first.name → starting or base address of the array name!

```
main()
{   int arr[30];
```

arr ⇒ starting address of this array

⇓

& arr[0]

arr[0] arr[18]

size of an integer variable is 4 bytes.

first.name[0]

```
struct number
{
    int a;
    char name[20];
} first, second;
```

→ tag name / type

first, second → instances / structure variable

```
struct number
{
    int a;
    char name[20];
    int *p;
} first, second;
```

4 Bytes
20 Bytes
4 Bytes
28 Bytes

```
main()
{
    int cost;
    ⋮
}
```

first.p = & cost;

struct number *q;

q → a

'*' → indirection operator

(*q).a

→ indirect membership operator
→ structure pointer operator

3-ways to access a structure member

1) using the structure name

first.a

2) (*q).a using a pointer to the structure with the indirection operator (*)

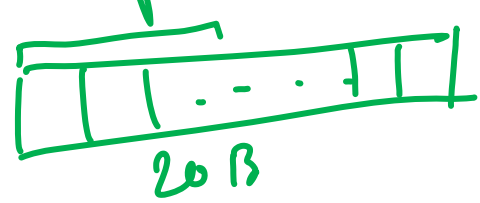
3) using a pointer to the structure with indirect membership operator

q→a

Structure vs Union

28 Bytes as per given eg.

20 Bytes as per given eg.



'typedef'

```
typedef struct
{   int a;
    char name[20];
} number;
```

structure identifier

number first, second ;

```
struct number
{
  . . .
  . . .
};

struct number first, second;
```

Self Referential Structure :—

```
struct record
{ int roll_no;
  char name[20];
  char branch[10];
  struct record *p;
}
```

a self-referential structure

points to a structure where it itself belongs.

```
struct record student1, student2;
```

p = &student1

"Using uninitialized pointer can be very dangerous. It can totally destroy a program"

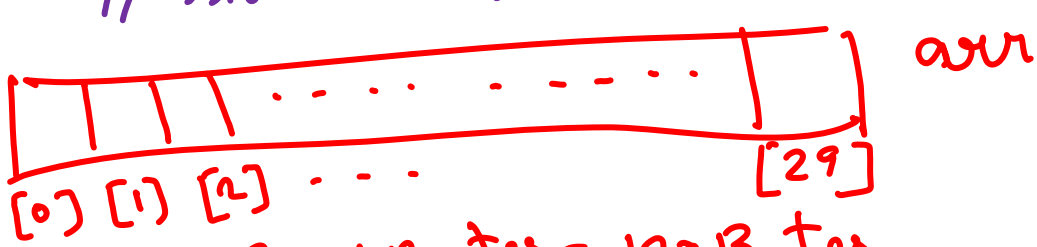
— X —

29/09/2021

Dynamic Memory Allocation

Space is allocated within the RAM and this allocation is conventionally done during the compilation time of an source code.

```
int arr[30];
:
// store only 10 integers
```



30 × 4 Bytes = 120 Bytes

So, this allocation of memory during compilation time, can remain redundant. This results in wastage of valuable memory space.

⇓

Static memory allocation

Dynamic Memory allocation: allotment of memory space is done run time or at the time of execution of the source code.

Static Allocation	Dynamic Allocation
1) We can have redundant memory space	1) No scope of redundant memory
2) Allocation is done during compilation time	2) Allocation is done during runtime
3) We don't need any particular memory allotting function.	3) We need specific memory allocating function.

4) Memory location allocated in static method are contiguous.	4) Memory locations are non-contiguous.

Dynamic Memory allocating function :-

1) malloc ( ) $\xrightarrow{\text{stored in}}$ stdlib.h

```
#include<stdlib.h>
:
main()
{
 :
}
```

allocates a memory space and returns the starting address of this memory.

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

(int *) → return type

sizeof(int) → number of bytes of memory which is to be allocated

2) calloc( )

2 parameters
- number of objects
- size of each of these objects

```
⋮
int *p;
scanf("%d", &num);
p = (int *) calloc(num, sizeof(int));
```

3) realloc( )

Dynamically allocated memory needs to be freed after its usage.

⇓

is also done by a function

free( )

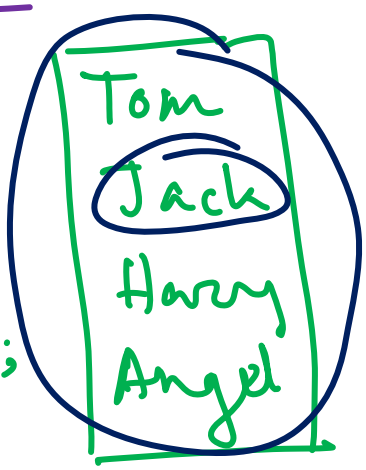
```
free(p);
```

⟹ the starting address of the dynamically allocated memory space which is freed.

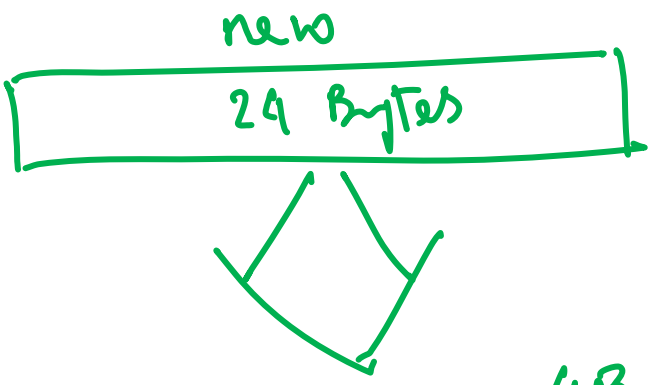
Self Referential Structures :-

```
struct person
{
    char name[20];
    struct person * next;
};
```

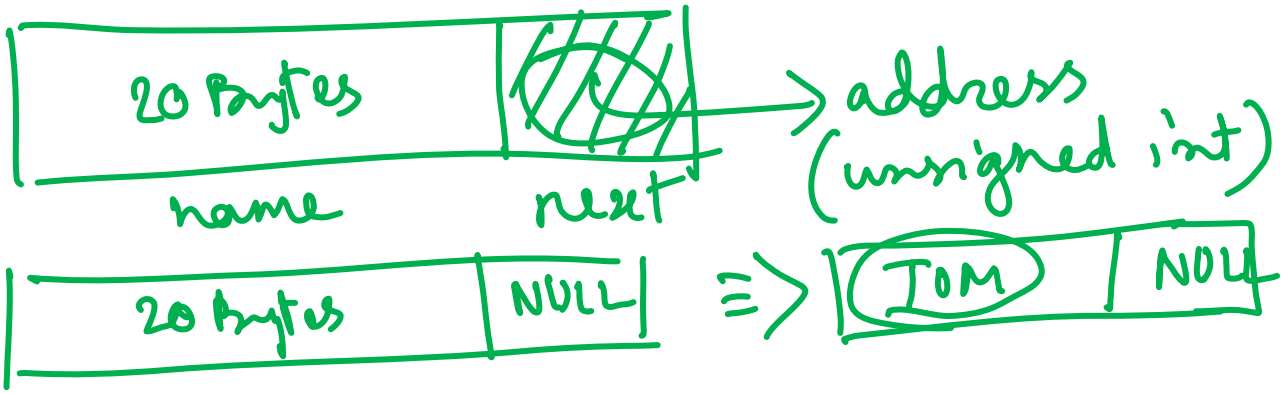
20 + 4
= 24 Bytes




```
struct person * new;
struct person * head;
new = (struct person *) malloc(sizeof(struct person));
head = NULL;
new -> next = head;
```

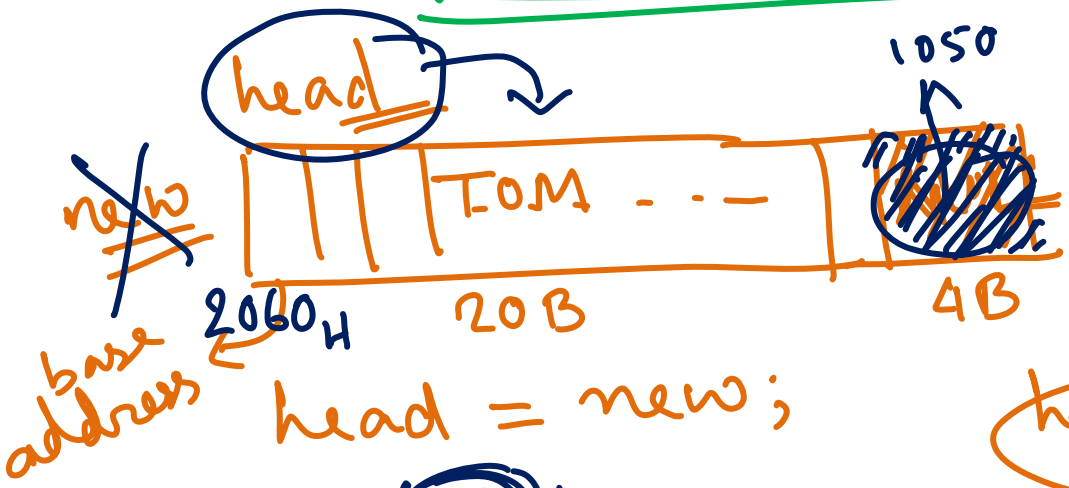


4B : Address of a structure



new → name

gets(new → name);

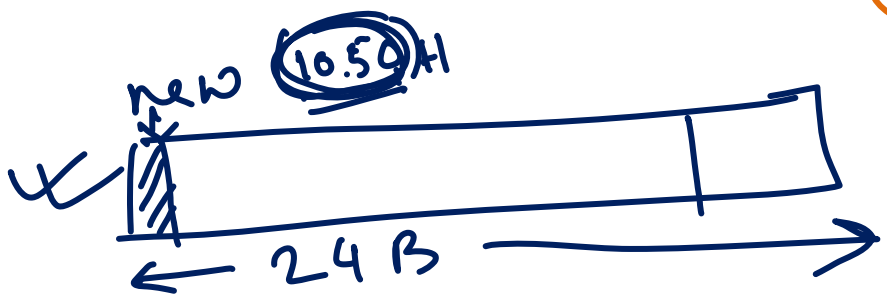


new stores the address of 0th byte

head = new;

head = NULL

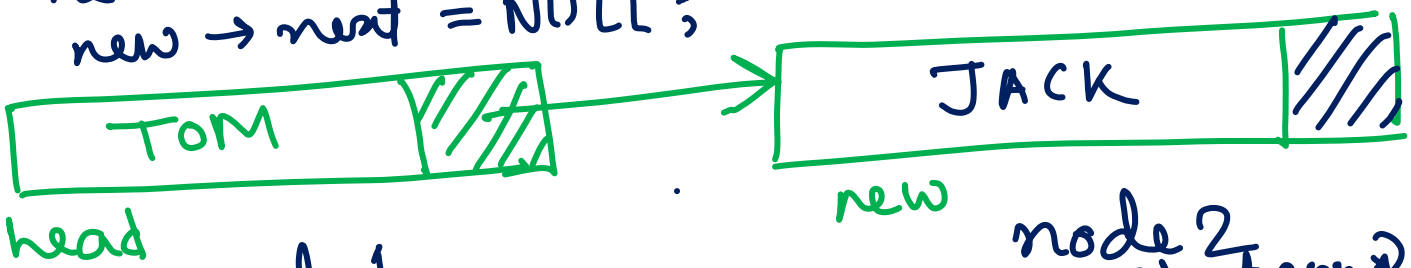
→ the address of the 0th byte



new = (struct person *) malloc(sizeof(struct person));

head → next = new;

new → next = NULL;



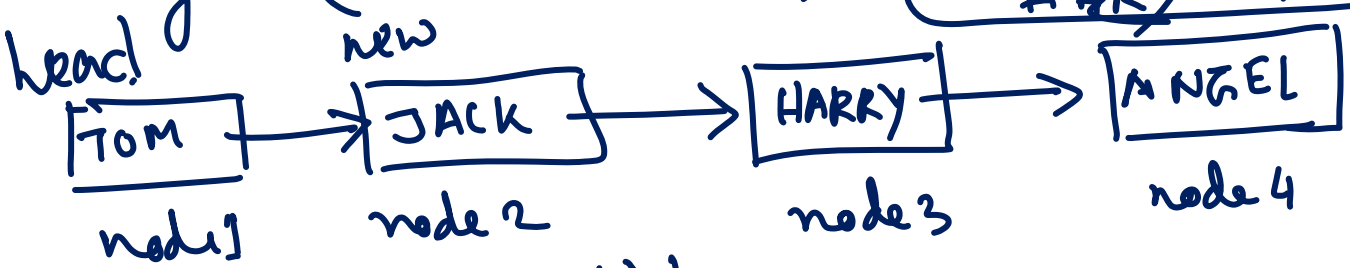
node 1

gets(new → name),

node 2

new2 = (struct person *) malloc

new → next = new2;



Linked List