unit 3 notes

STRINGS

- string in C is an array of characters and terminated with a special character ‘\0’
- char ch = 'x' ; // character constant in single quote. always of 1 byte
- char ch[ ] = "x" ; // String constant. always in double quotes. Terminated with '\0'. Always 1 byte more when specified between “ and ” in initialization.
  Note: When the compiler encounters a sequence of characters enclosed in the double quotes, it appends a null character ‘\0’ at the end by default

Initialization:

If the size is not specified, the compiler counts the number of elements in the array and allocates those many bytes to an array.

If the size is specified, it allocates those many bytes and unused memory locations are initialized with default value '\0'. This is partial initialization.

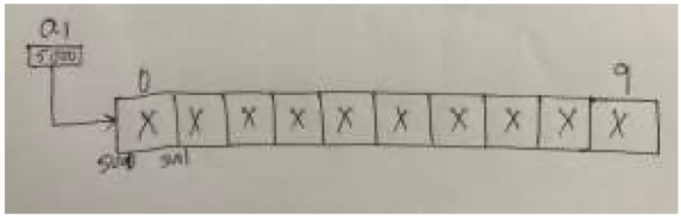
If the string is hard coded, it is programmer’s responsibility to end the string with '\0'character.

```
char a[] = { 'C', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm, 'i', 'n', 'g', '\0' };
char b[] = "C Programming" ;
```

basics

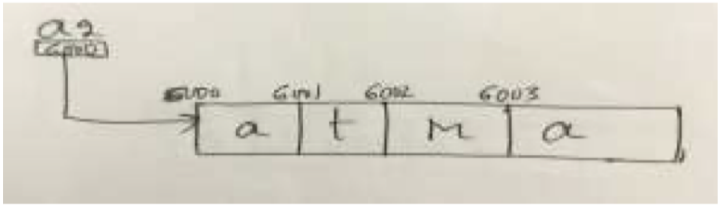
code examples:

```
char a1[10];
// Declaration: Memory locations are filled with undefined values/garbage value
printf("sizeof a1 is %d",sizeof(a1));                // 10
```



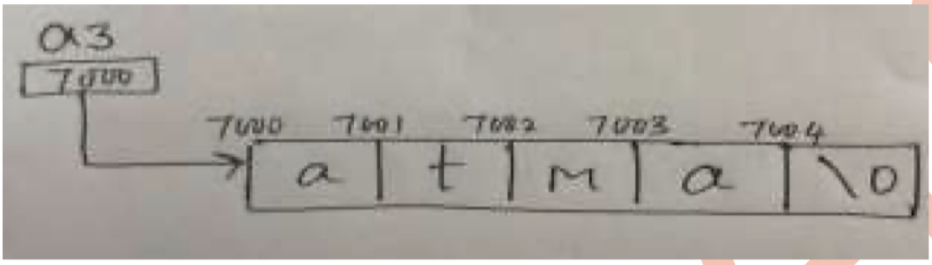
char a2[ ] = {'a','t','m','a'};//**Initialization**

printf("sizeof a2 is %d",sizeof(a2)); // 4 but cannot assure about a2 while printing



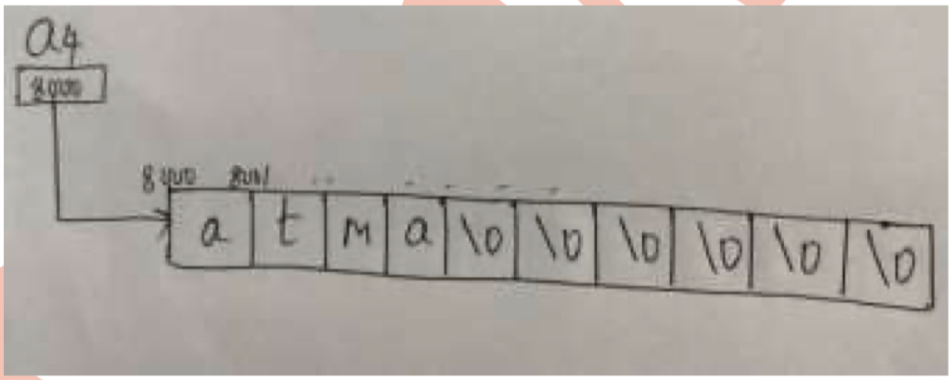
char a3[ ] = "atma" ;

printf("sizeof a3 is %d",sizeof(a3)); // 5sure about a3 while printing

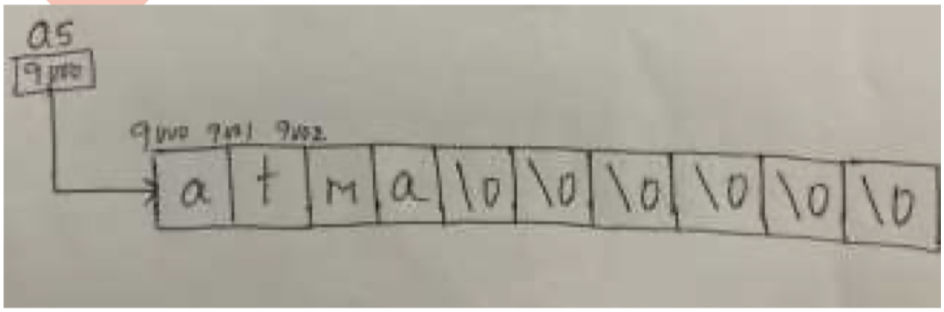


char a4[10] = {'a','t','m','a'}// **Partial Initialization**

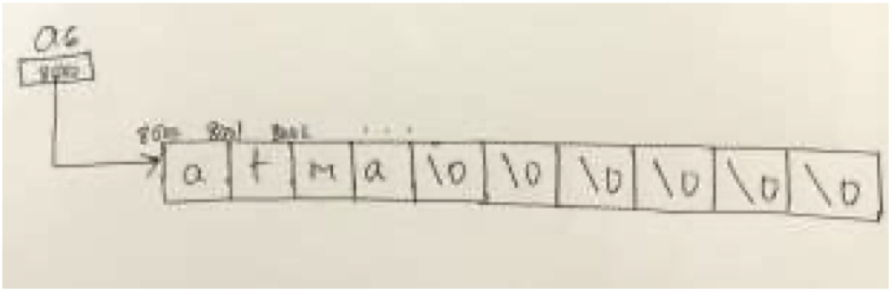
printf("sizeof a4 is %d",sizeof(a4));// 10sure about a4 while printing



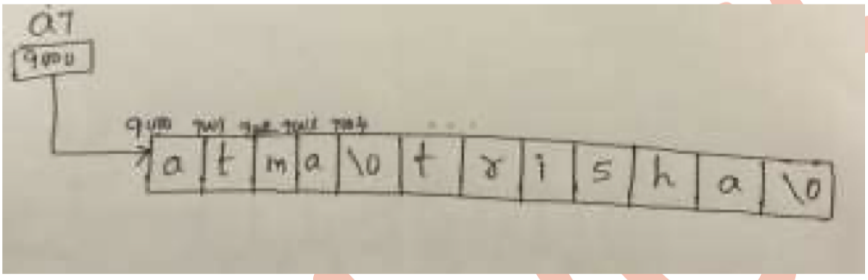
```
char a5[10 ] = "atma" ;
printf("sizeof a5 is %d",sizeof(a5));// 10 sure about a5 while printing
```



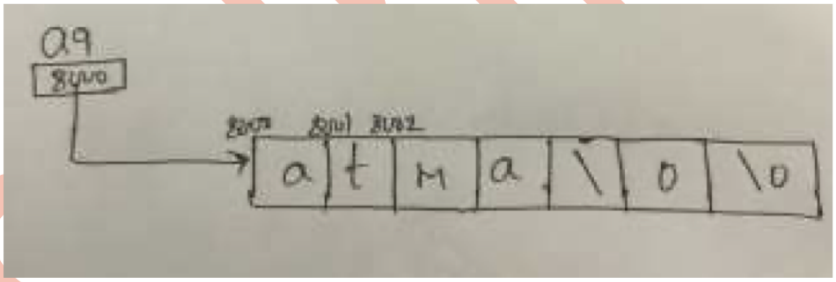

```
char a6[10 ] = {'a','t','m','a', '\0'};
printf("sizeof a6 is %d",sizeof(a6));// 10 sure about a6 while printing
```



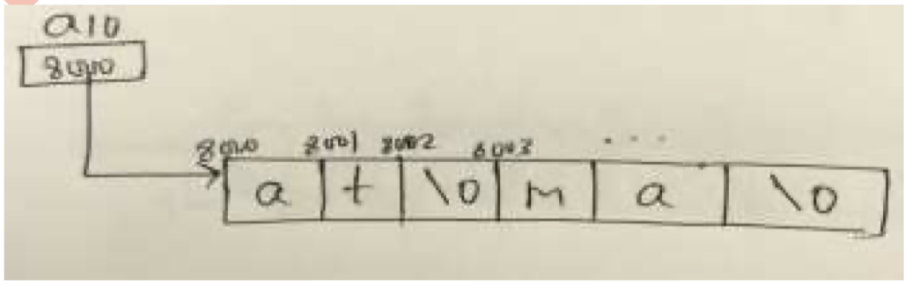

```
char a7[ ] = {'a', 't', 'm', 'a', '\0', 't', 'r', 'i', 's', 'h', 'a', '\0' };
printf("sizeof a7 is %d",sizeof(a7));  // 12 a7 will be printed only till first '\0'
```



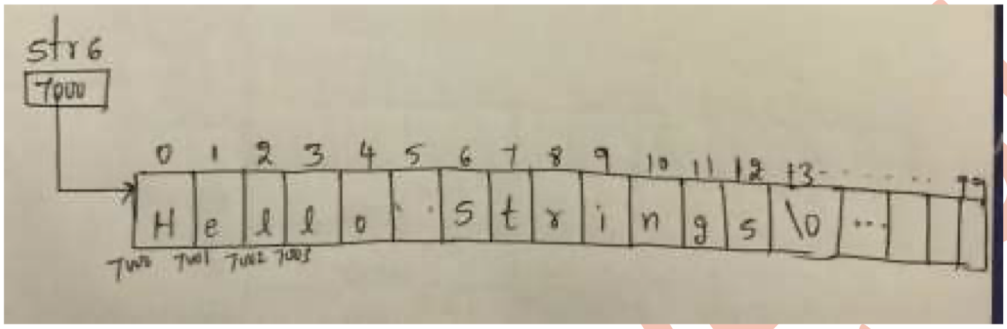
```
char a9[ ] = "atma\\0" ;
printf("sizeof a9 is %d",sizeof(a9));// 7 sure about a9 while printing
```




```
char a10[ ] = "at\0ma" ;
printf("sizeof a10 is %d",sizeof(a10));//6 a10 will be printed only till first '\0'
```

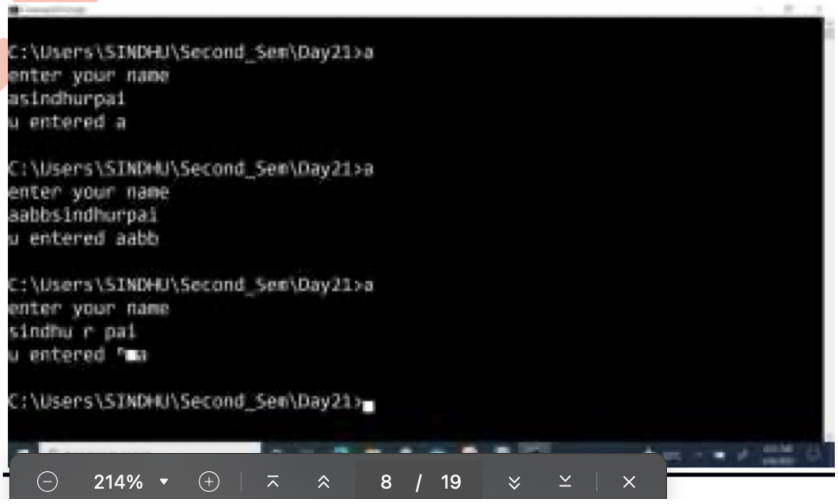



```
char str6[100] ;
printf("Enter the string");
scanf(("%[^\n]s", str6);                    // User entered Hello Strings and pressed enter key
printf("%s", str6);                         // Hello Strings
```



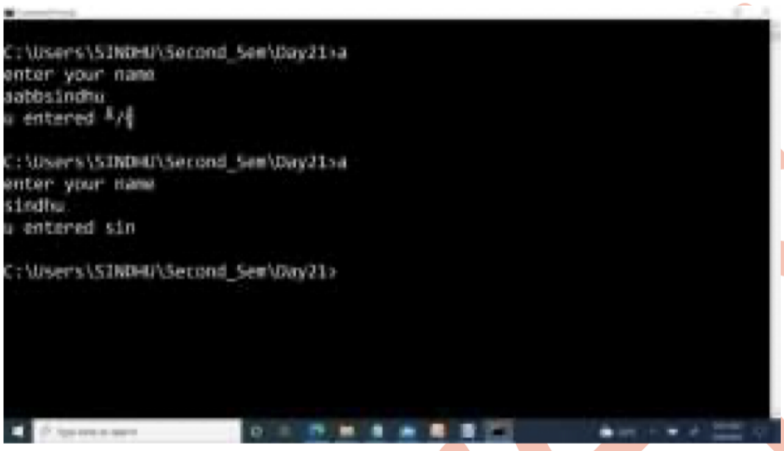
```
char str6[100] ;
printf("enter your name\n");
scanf("%[abcd]s",str6); // [ ] chracter class, starting with either a or b or c or d.
//When it encounters other characters, scanf terminates
printf("u entered %s\n",str6);
```

Output:




```
char str7[100] ;
printf("enter your name\n");
scanf("%[^abcd]s",str6); // neither a nor b nor c nor d
//When it encounters any of these characters, scanf terminates
printf("u entered %s\n",str6);
```

Output:



pointer vs strings

```
char *p1 = "pesu";
//p1 is stored at stack.  "pes" is stored at code segment of memory. It is read only.
printf("size is %d\n", sizeof(p1));                    // size of pointer
// This statement assigns to p variable a pointer to the character array
printf("p1 is %s", p1) ;          //pesu   p1 is an address. %s prints until \0 is encountered
p++;    // Pointer may be modified to point elsewhere.
printf("p1 is %s", p1) ;          //esu
p1[1] = 'S' ;     // No compile time Error
printf("p1 is %s", p1) ; //Behaviour is undefined if you try to modify the string contents


char p2[] = "pesu";   // Stored in the Stack segment of memory
printf("size is %d\n", sizeof(p2));      // 5 = number of elements in array+1 for '\0'
printf("p2 is %s", p2) ;          //pesu
p2[1] = 'E';     //Individual characters within the array may be modified.
printf("p2 is %s", p2) ;          //pEsu
p2++;  // Compile time Error
// Array always refers to the same storage. So array is a Constant pointer
```

Built-in String Functions

- **strlen(a)** – Expects string as an argument and returns the length of the string, excluding the NULL character
- **strcpy(a,b)** – Expects two strings and copies the value of b to a.
- **strcat(a,b)** – Expects two strings and concatenated result is copied back to a.
- **strchr(a,ch)** – Expects string and a character to be found in string. Returns the address of the matched character in a given string if found. Else, NULL is returned.
- **strcmp(a,b)** – Compares whether content of array a is same as array b. If a==b, returns 0. Returns +ve, if array a has lexicographically higher value than b. Else, -ve.

code 1:

```
#include<stdio.h>

#include<string.h>

int main(){

char str1[]= "pes";

printf("string length is %d\n",strlen(str1));

// 3

char a[10]="abcd";

char b[20],c[10];

//b=a;

//we are trying to equate two addresses. Array Assignment incompatible

strcpy(b,a);

// copy a to b.

printf("a is %s and b is %s\n",a,b);
```

```
printf("length of a is %d and length of b is %d\n",strlen(a),strlen(b));

return 0;

}
```

output:

```
string length is 3
a is abcd and b is abcd
length of a is 4 and length of b is 4
```

code 2:Comparing two strings using Lexicographical ordering

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

int main() {
    char a[10] = "abcd";
    char b[10] = "abcd";
    char c[10] = "AbCD";
    char d[10] = "abD";

    int i = strcmp(a, b);
    int j = strcmpi(a, c); // ignore case
    int k = strncmp(a, d, 2); // compare only n characters, here 2

    printf("first %d--cmp\t%d--cmpi\t%d--ncmp\n", i, j, k);

    i = strcmp(a, d);
    j = strcmpi(a, d);
    k = strncmp(a, d, 3);

    printf("later %d--cmp\t%d--cmpi\t%d--ncmp\n", i, j, k);

    return 0;
}
```

output:

```
first 0--cmp    0--cmpi 0--ncmp
later -1--cmp   0--cmpi -1--ncmp
```

code 3:

```
#include <stdio.h>

#include <string.h>

int main() {

char a[100]="abcd";

char b[100]="pqr";

char c[100]="whatsapp";

strcat(a,b);

printf("a is %s and b is %s\n",a,b);

printf("length of a is %d and length of b is %d\n",strlen(a),strlen(b));

printf("a is %s and size is %d\n",a,strlen(a));

return 0;

}
```

output:

```
a is abcdpqr and b is pqr
length of a is 7 and length of b is 3
a is abcdpqr and size is 7
```

code4:

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

int main() {
    char ch[] = "This is a test";
    char *pos = strchr(ch, 't'); // Find the first occurrence of 't' in the
string
    int p = pos - ch; // Calculate the position of 't' by subtracting the
address of 't' from the address of the beginning of the string

    if (pos) {
        printf("Character 't' found at position %d\n", p);
    } else {
        printf("Character 't' not found\n");
    }

    return 0;
}
```

output:

```
Character 't' found at position 0
```

String operations using User defined functions

to find length of string without using built in function

```
#include <stdio.h>

#include <string.h>



int length(char str[]);
```

```
int main() {

char str1[]="hello";

int count=length(str1);

printf("%d",count);

return 0;

}

int length(char str[]){

int count=0;

for(int i=0;str[i]!='\0';i++){

count++;

}

return count;

}
```

output:

```
5
```

string copy without using built in function

```
#include <stdio.h>

#include <string.h>
```

```
void copy(char str1[],char str2[]);



int main() {

char str1[]="hello";

char str2[]="";

copy(str1,str2);

return 0;

}

void copy(char str1[],char str2[]){

int i;

for(i=0;str1[i]!='\0';i++){

str2[i]=str1[i];

}

str2[i]='\0';

printf("copied string is %s",str2);

}
```

output:

```
copied string is hello
```

wap for string compare

code:

```
#include <stdio.h>

#include <string.h>



int compare(char str1[],char str2[]);



int main() {

char str1[]="hello";

char str2[]="Hello";

int p=compare(str1,str2);

if(p==strlen(str1))

printf("both are same");

else

printf("not same");

return 0;

}

int compare(char str1[],char str2[]){

int i;

int count;

for(i=0;str1[i]!='\0';i++){

if(str2[i]==str1[i])
```

```
count++;

}

return count;

}
```

output:

```
not same
```

wap to search a character

code:

```
#include <stdio.h>

#include <string.h>



int search(char str1[],char ch);



int main() {

char str1[]="hello";

char ch='e';

int p=search(str1,ch);

if(p>0)

printf("%d",p);

else
```

```
printf("no");

return 0;

}

int search(char str1[],char ch){

int i;

int count;

for(i=0;str1[i]!='\0';i++){

if(ch==str1[i])

count++;

break;

}

return count;

}
```

output:

```
1
```

note:SEE DIFFERENT VERSIONS OF IMPLEMENTATION IN NOTES PDF

MEMORY ALLOCATION

Memory can be allocated for variables using different techniques.

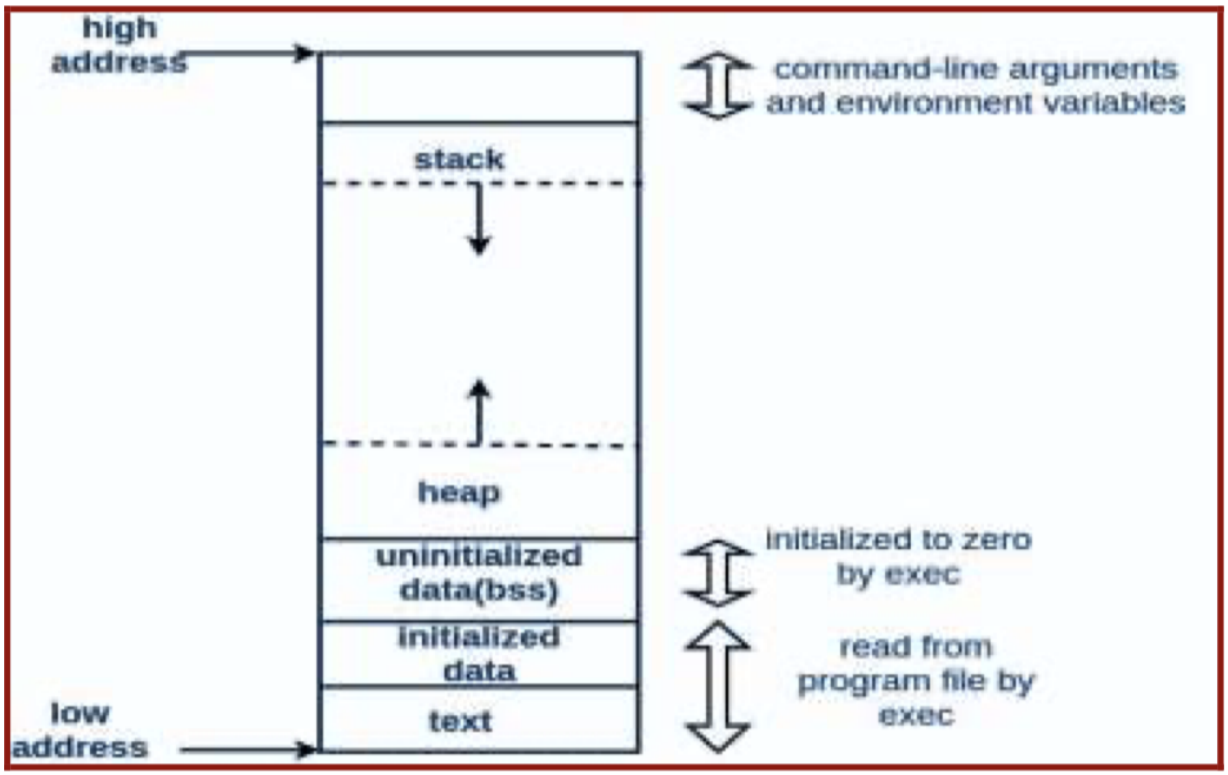
1. Static allocation
   decided by the compiler allocation at load time before the execution or run time

2. Automatic allocation
   decided by the compiler allocation at run time
   allocation on entry to the block and deallocation on exit
3. Dynamic allocation
   code generated by the compiler allocation and deallocation on call to memory allocation functions: malloc, calloc, realloc and deallocation function: free

Dynamic Allocation of Memory -allocating memory at runtime/execution time

NOTE

- When the memory is allocated during compile-time it is stored in the Static Memory and it is known as Static Memory Allocation
- when the memory is allocated during run-time, it is stored in the Dynamic Memory and it is known as Dynamic Memory Allocation.



Text segment

It contains **machine code** of the **compiled program**. **Usually**, it is **sharable** so that **only** a **single copy** needs to be in **memory** for **frequently** executed **programs**, such as **text editors**, the **C compiler**, the **shells**, and **so on**. The **text segment** of an **executable object file** is usually **read-only segment** that **prevents** a **program** from being **accidentally modified.**

Initialized Data Segment

Stores all **global, static, constant, and external variables** - declared with extern keyword that are initialized beforehand. It is **not read-only,** since the values of the variables can be changed at run time. This segment can be further classified into initialized read-only area and initialized read-write area.

```
#include <stdio.h>
const char Entity[]="PES University";   /* global variable stored in Initialized Data Segment in read-only area*/
char Faculty[]="Nitin V Pujari";     /*  global variable stored in Initialized Data Segment in read-write area*/
int main()
{
    static int Value = 11;          /* static variable stored in Initialized Data Segment*/
    return 0;
}
```

Uninitialized Data Segment(bss)

Data in this segment is initialized to arithmetic 0 before execution of the program. Uninitialized data starts at the end of the data segment and contains all **global variables and static variables that are initialized to 0** or **do not have explicit initialization in source code.**

```
#include <stdio.h>
char Entity;   /* Uninitialized variable stored in bss*/
char Faculty;  /* Uninitialized variable stored in bss*/
int main()
{
    static int Value; /* Uninitialized static variable stored in bss */
    return 0;
}
```

Heap Segment

It is the segment where dynamic memory allocation usually takes place. When some more memory need to be allocated using malloc and calloc function, heap grows upward. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

```
#include <stdio.h>

int main()
{
    int *Value=(int*)malloc(sizeof(int)); /* memory allocating in heap segment */
    return 0;
}
```

Stack Segment

Is used to **store all local variables and is used for passing arguments to the functions** along with the return address of the instruction which is to be executed after the function call is completed.Local variables have a scope to the block where they are defined in, they are created when control enters into the block. All recursive function calls are added to stack.

Note: The stack and heap are traditionally located at opposite ends of the process's virtual address space

There is no operator in C to support dynamic memory management. Library routines known as "Memory management functions" are used for allocating and freeing/releasing memory during execution of a program. These functions are defined in stdlib.h

malloc():

1. **Allocates requested size of bytes**: When you call `malloc(size)`, you're asking the computer to allocate a contiguous block of memory of a certain size, specified by the `size` parameter, in bytes.
2. **Returns a void pointer pointing to the first byte of the allocated space**: After allocating the memory, `malloc()` returns a void pointer (`void *`) which points to the starting address of the allocated memory block. Since `malloc()` doesn't know what type of data you're going to store in the memory block, it returns a generic pointer type (`void *`).
3. **The memory is not initialized**: It's important to note that the memory returned by `malloc()` is not initialized with any specific values. The contents of the allocated memory block are unpredictable and may contain arbitrary data leftover from previous usage.
4. **Suitably aligned for any kind of variable**: The memory block returned by `malloc()` is aligned in a way that it can accommodate any kind of data variable. This means you can safely use it to store variables of different types like integers, floats, characters, etc.
5. **On error, returns NULL**: If `malloc()` encounters an error during memory allocation, such as running out of available memory, it returns a `NULL` pointer to indicate failure.
6. **NULL may also be returned by a successful call to malloc() with a size of zero**: Interestingly, if you call `malloc(0)`, it may still return `NULL`. This behavior is implementation-dependent and is used as a convention to handle edge cases.

Prototype: void *malloc(size_t size);

EXAMPLES:

code 1:

```
#include<stdio.h>

#include<stdlib.h>

int main(){

int* p = (int*)malloc(sizeof(int)); //dynamic allocation for one integer

//address of that location is returned in p

*p = 10;

printf("p = %d", *p);

return 0;

}
```

output

```
p=10
```

code 2:

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

int main() {
    int *x;
    int n, i;
    printf("Enter the number of elements\n");
    scanf("%d", &n);
    x = (int*)malloc(n * sizeof(int)); // memory will be allocated for n
integer numbers
    if (x == NULL) // to check if memory is allocated or not by malloc
        printf("Memory not allocated.\n");
```

```
    else {
        printf("Memory successfully allocated using malloc.\n");
        printf("Enter the integer values:\n ");
        for (i = 0; i < n; i++) // To read inputs from user
        {
            scanf("%d", &x[i]);
        }
        printf("Output: "); // Printing
        for (i = 0; i < n; i++)
            printf("%d\t", *(x + i));
    }
    return 0;
}
```

output:

```
Enter the number of elements
3
Memory successfully allocated using malloc.
Enter the integer values:
5
6
7
Output: 5       6       7
```

calloc():

1. **Allocates space for elements, initializes them to zero**: Unlike `malloc()`, which doesn't initialize the allocated memory, `calloc()` initializes the memory it allocates to zero. This means that when you allocate memory using `calloc()`, you can be sure that all elements in the allocated block will be set to zero.
2. **Returns a void pointer to the memory**: Like `malloc()`, `calloc()` also returns a void pointer (`void *`) to the allocated memory block. Since it doesn't know what type of data you're going to store in the memory block, it returns a generic pointer type.
3. **Prototype: void *calloc(size_t nmemb, size_t size)**: The `calloc()` function prototype takes two parameters: `nmemb` and `size`. `nmemb` specifies the number of elements to allocate memory for, while `size` specifies the size of each element in bytes.
4. **The calloc() function allocates memory for an array of nmemb elements of size bytes each**: This means that `calloc()` calculates the total amount of memory required by multiplying `nmemb` with `size`, and then allocates that amount of memory.
5. **The `calloc()` function returns a pointer to the allocated memory that is suitably aligned for any kind of variable**: Just like `malloc()`, the memory allocated by `calloc()` is aligned in a way that it can accommodate any kind of data variable.
6. **On error, this function returns NULL**: Similar to `malloc()`, if `calloc()` encounters an error during memory allocation, such as running out of available memory, it returns a `NULL` pointer to indicate failure.
7. **NULL may also be returned by a successful call to calloc() with nmemb or size equal to zero**: Similar to `malloc()`, if you call `calloc()` with either `nmemb` or `size` equal to zero, it may return `NULL`. This behavior is used as a convention to hand dge cases.

code 1:

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

int main() {
    int *x;
    int n, i;

    printf("Enter the number of elements\n");
    scanf("%d", &n);

    x = (int*)calloc(n, sizeof(int)); // a block of n integers (array) is
created dynamically
```

```
    if (x == NULL) // to check if memory is allocated or not
        printf("Memory not allocated.\n");
    else {
        printf("Memory successfully allocated using calloc.\n");
        printf("Enter the integer values:\n ");

        for (i = 0; i < n; i++) { // To read inputs from user
            scanf("%d", &x[i]);
        }

        printf("Output: "); // Printing

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

    return 0;
}
```

output:

```
Enter the number of elements
5
Memory successfully allocated using calloc.
Enter the integer values:
10
20
30
40
50
Output: 10      20      30      40      50
```

realloc():

Purpose:

The `realloc()` function is used to modify the size of a previously allocated memory block. It can increase or decrease the size of the block and optionally copy the contents of the old block to the new block.

Prototype:

c Copy code

```c
void *realloc(void *ptr, size_t size);
```

Parameters:

- `ptr`: Pointer to the previously allocated memory block that you want to resize.
- `size`: The new size, in bytes, that you want the memory block to be resized to.

Behavior:

- If `ptr` is `NULL`, the behavior of `realloc()` is the same as calling `malloc(size)`.
- If `size` is zero, the behavior of `realloc()` is the same as calling `free(ptr)` and the function returns `NULL`.
- If the new size is smaller than the old size, the excess memory beyond the new size is freed, and the content of the memory block is preserved up to the new size.
- If the new size is larger than the old size, a new memory block is allocated, and the content of the old block is copied to the new block up to the lesser of the new and old sizes. The added memory beyond the old size is uninitialized.
- If `realloc()` fails to allocate memory for the new block, it returns `NULL`, and the old memory block remains unchanged.

Return Value:

- If the reallocation is successful, `realloc()` returns a pointer to the newly allocated memory block.
- If the reallocation fails, `realloc()` returns `NULL`, and the original memory block remains unchanged.

code 1:

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

int main() {
    int *p = (int*)malloc(3 * sizeof(int));
```

```
    if (p == NULL) {
        printf("Memory not allocated\n");
        return 1;
    } else {
        int i;
        printf("Initial address is %p\n", p);
        printf("Enter three elements:\n");
        for (i = 0; i < 3; i++) {
            scanf("%d", &p[i]);
        }
        printf("Entered elements are:\n");
        for (i = 0; i < 3; i++) {
            printf("%d\t", p[i]);
        }
        printf("\nEnter the new size:\n");
        int new_size;
        scanf("%d", &new_size);
        int *new_p = (int*)realloc(p, new_size * sizeof(int));
        if (new_p == NULL) {
            printf("Memory not allocated\n");
            free(p); // Free the previously allocated memory
            return 1;
        } else {
            p = new_p;
            printf("\nNew address is %p\n", p);
            printf("Enter %d new elements:\n", new_size - 3);
            for (i = 3; i < new_size; i++) {
                scanf("%d", &p[i]);
            }
            printf("New elements are:\n");
            for (i = 0; i < new_size; i++) {
                printf("%d\t", p[i]);
            }
        }
    }
    free(p); // Free the allocated memory before exiting
    return 0;
}
```

output:

```
Initial address is 0x12af04170
Enter three elements:
5
3
4
Entered elements are:
5       3       4
Enter the new size:
5

New address is 0x12af04240
Enter 2 new elements:
3
5
New elements are:
5       3       4       3       5
```

free():

Releases the allocated memory and returns it back to heap. Must be applied with malloc(), calloc() or realloc()

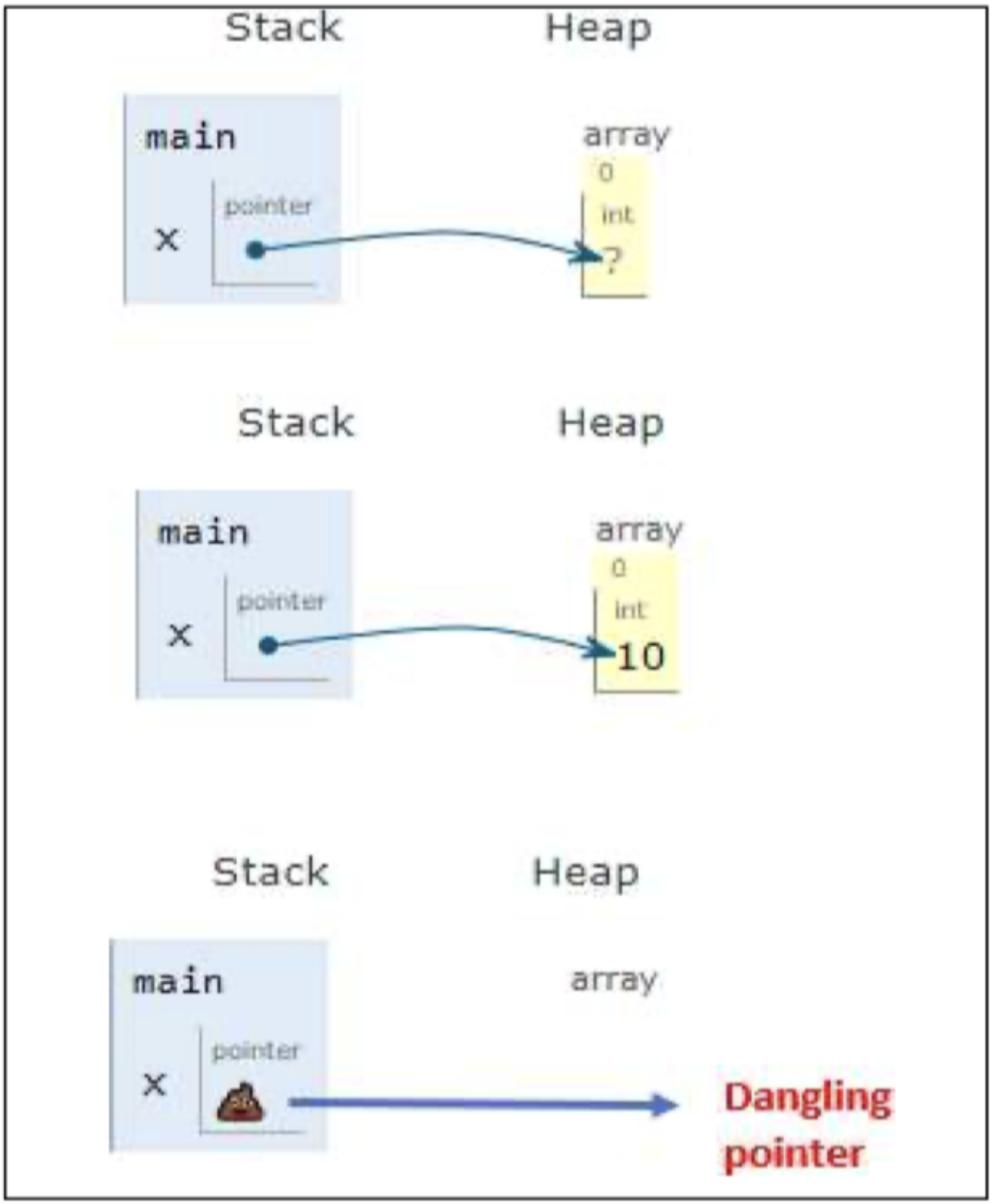
code 1:

```
#include <stdlib.h>

int main() {

int *x;

// Allocate memory for one integer

x = (int*)malloc(sizeof(int));

if (x == NULL) {
```

```
printf("Memory allocation failed\n");

return 1;

}

// Store value in the allocated memory

*x = 10;

printf("x = %p\n", (void *)x);

printf("x is pointing to = %d\n", *x);

// Free the allocated memory

free(x);

printf("Memory successfully freed\n");

printf("x is pointing to = %d\n", *x);

return 0;

}
```

output:

```
x = 0x133605f80
x is pointing to = 10
Memory successfully freed
x is pointing to = 0
```



How does free function knows how much memory must be returned/released?
On allocation of Memory using memory management functions, it stores somewhere in memory the # of bytes allocated. This is known as book keeping information. We cannot access this info. But implementation has access to it. The function free finds this size given the pointer returned by allocation functions and de-allocates the required amount of memory.
By observing both the diagrams above, we can get to know that block allocated using functions will be deallocated and hence the pointer becomes dangling.

COMMON ERRORS DURING MEMORY ALLOCATIONS:

dangling pointer:

A pointer which points to a location that doesn’t exist is known as dangling pointer. It can happen anywhere in the memory segment. Solution is to assign the pointer to NULL. Situations that
results in dangling pointer are as below.
• Freeing the memory results in dangling pointer
• Using new pointer variable to store return address in realloc results in dangling pointer Note: Dereferencing the dangling pointer results in undefined behavior

code 1:

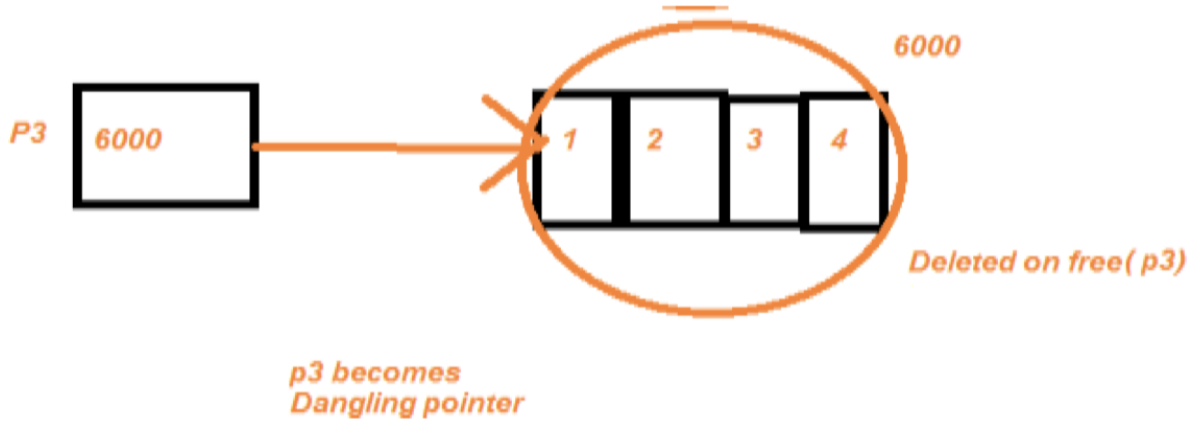
```
#include<stdio.h>
#include<stdlib.h>

int main() {
    // Allocate memory for one integer
    int *p2 = (int*)malloc(sizeof(int));
    // conversion from void* to int*
    printf("%p\n", p2); // Print the address of the allocated memory
    *p2 = 200; // Store the value 200 in the allocated memory
    printf("%p\n", p2); // Print the address of the allocated memory (should
be the same)
    printf("%d\n", *p2); // Print the value stored in the allocated memory
(should be 200)
    free(p2); // Free the allocated memory
    return 0;
}
```

output:

```
0x130e05f80
0x130e05f80
200
```

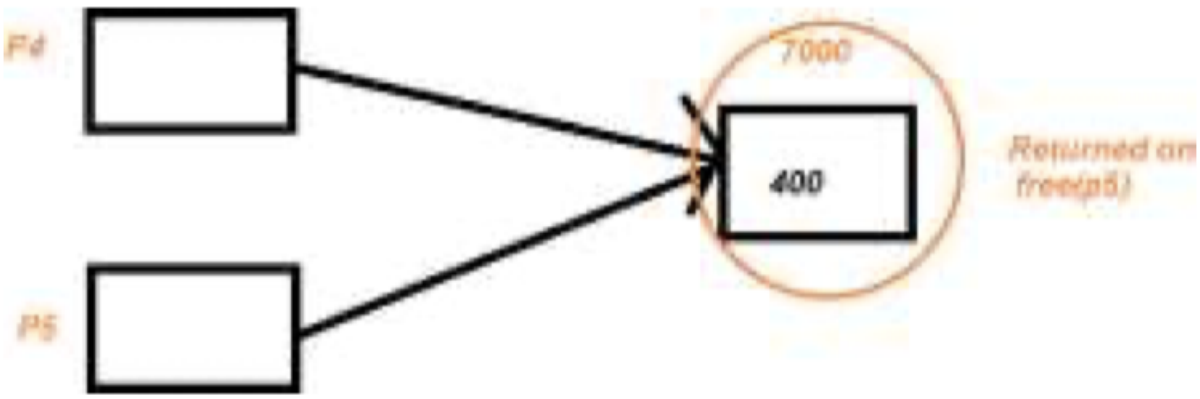
code 2:

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

int main() {
    printf("Enter the number of integers you want to store\n");
    int n;
    int i;
    scanf("%d", &n); // Read the number of integers from the user

    // Allocate memory for n integers
    int *p3 = (int*)malloc(n * sizeof(int)); // Initially all values are
undefined

    printf("Enter %d elements\n", n);
    for(i = 0; i < n; i++)
        scanf("%d", &p3[i]); // Read integer values for each element from
the user

    printf("Entered elements are\n");
    for(i = 0; i < n; i++)
        printf("%d\n", p3[i]); // Print the entered elements

    free(p3); // Free the allocated memory
    return 0;
}
```



code 3:

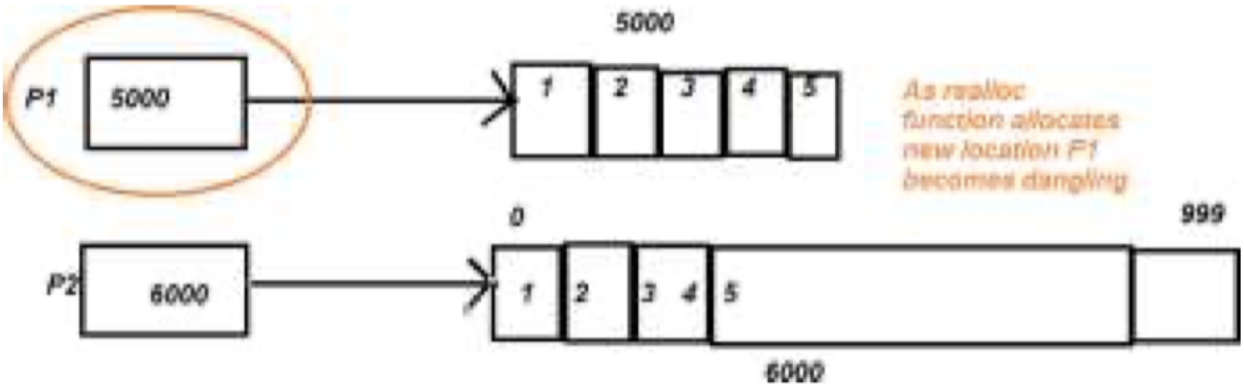
```
int *p4 = (int*)malloc(sizeof(int)); // Allocate memory for one integer
*p4 = 400; // Store the value 400 in the allocated memory
printf(" %d\n", *p4); // Print the value stored in the allocated memory
int *p5 = p4; // Assign the pointer p4 to the pointer p5
free(p5); // Free the allocated memory
```



code 4:

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

int main() {
    int *p1 = (int *) calloc(5, sizeof(int)); // Allocate memory for 5
integers and initialize to 0
    printf("before %p\n", p1); // Print the address of the allocated memory
block
    printf("enter the elements\n");
    for(int i = 0; i < 5; i++)
        scanf("%d", &p1[i]); // Read input for each integer
    for(int i = 0; i < 5; i++)
        printf("%d ", p1[i]); // Print the entered integers

    // Reallocate memory for 1000 integers
```

```
    int *p2 = (int*) realloc(p1, 1000 * sizeof(int)); // p1 becomes dangling
if new locations are allotted
    printf("after increasing size %p\n", p2); // Print the address of the
reallocated memory block
    // Same address as p1 if the same location can be extended, else
different address
    printf("content at address pointed by p2 = %d\n", *p2); // Print the
first element of the reallocated memory block
    printf("after realloc %p\n", p1); // Print the address pointed to by p1
(may be different after realloc)
    printf("contents at address pointed by p1 = %d\n", *p1); // Print the
first element of the original memory block
    return 0;
}
```



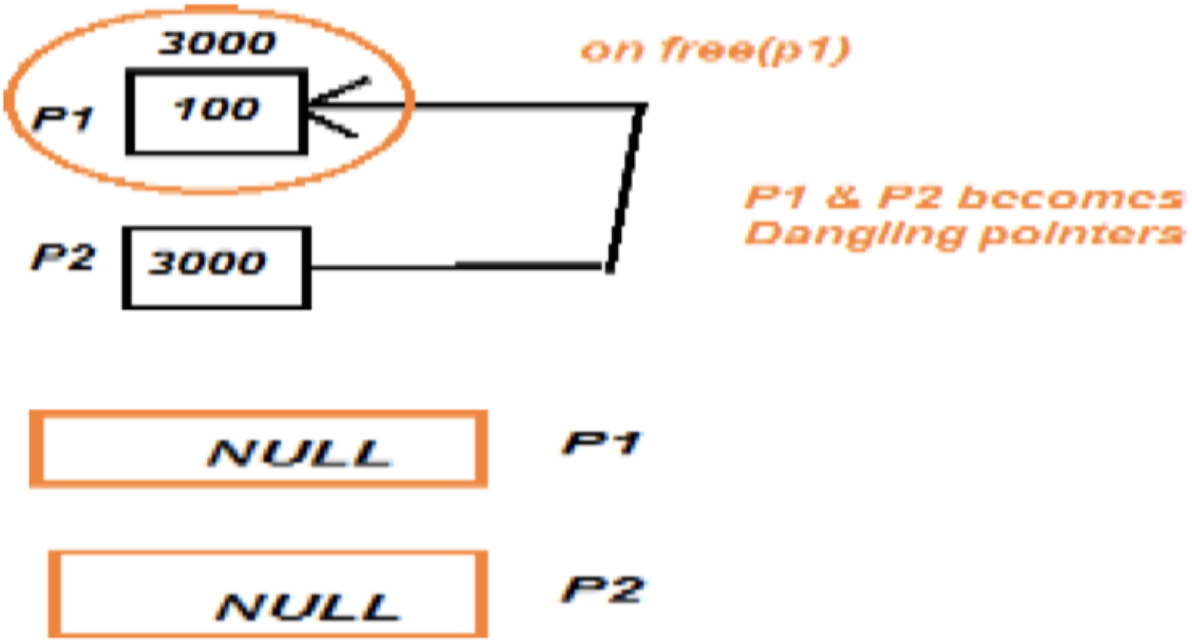
null pointer:

It is always a good practice to assign the pointer to NULL once after the usage of free on the pointer to avoid dangling pointers. This results in NULL Pointer. Dereferencing the NULL pointer results in guaranteed crash

code 1:

```
int *p1 = (int*)malloc(sizeof(int));
*p1 = 100;
printf(" %d\n", *p1); // 100 int
*p2 = p1;
```

```
printf(" %d\n", *p2); // 100
free(p1); // p1 and p2 both becomes dangling pointer.
p1 = NULL; // safe programming
p2 = NULL;
printf(" %d\n", *p1);// Guaranteed crash
printf(" %d\n", *p2);
```



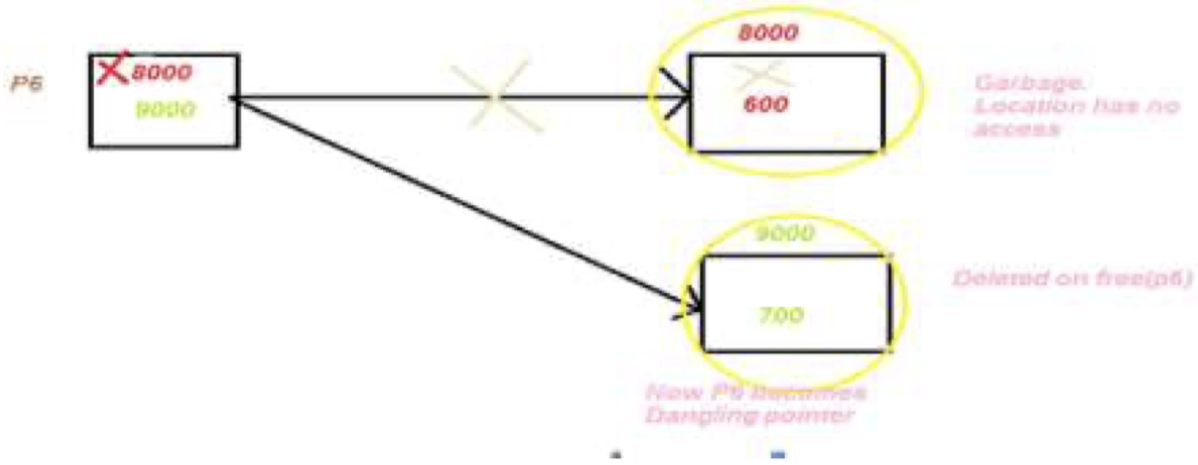
garbage and memory leakage

Garbage is a location which has no name and hence no access. If the same pointer is allocated memory more than once using the DMA functions, initially allocated spaces becomes garbage. Garbage in turn results in memory leak. Memory leak can happen only in Heap region.

code 1:

```
#include<stdio.h>
#include<stdlib.h>
int main() {

int *p6 = (int*)malloc(sizeof(int));
*p6 = 600;
printf(" %d\n", *p6); // 600
```

```
p6 = (int*)malloc(sizeof(int));
// changing p6 loses the pointer to the
//location allocated by the previous malloc
// So memory allocated by previous malloc has no access. Hence becomes
garbage *p6 = 700;
printf(" %d\n", *p6);
free(p6);
return 0;

}
```



double free error(DO NOT TRY THIS)

• If free() is used on a memory that is already freed before
• Leads to undefined behavior.
• Might corrupt the state of the memory manager that can cause existing blocks of memory to get corrupted or future allocations to fail.
• Can cause the program to crash or alter the execution flow

```
int* p = (int*)malloc(sizeof(int));
*p = 10;
printf("p = %d", *p);
free(p);
free(p);
The above code might lead to undefined behavior.  So, it is a good practice
to make the pointer NULL after the usage of free. By chance, if the pointer
is freed again, it doesn't do anything with NULL. free(p);
```

```
p = NULL; //Function free does nothing with a NULL pointer.
free(p);
```

programs on strings

Program to find the count of given character in a given string.

```
#include<stdio.h>

#include<string.h>

int main(){

char str1[100];

scanf("%s",str1);

int n=strlen(str1);

char ch='l';

int count=0;

for(int i=0;i<n;i++){

if(str1[i]==ch){

count++;

}

}

printf("%d",count);

return 0;

}
```

output:

```
hello
2
```

different version:

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

// Function declaration
int find_frequency(char *s, char ch);

int main() {
    char str1[100];
    printf("Enter the string: ");
    scanf("%[^\n]s", str1); // Read the string input (including spaces)

    printf("Enter the character: ");
    fflush(stdin);
    char ch = getchar(); // Read the character input

    int count = find_frequency(str1, ch);
    printf("%c is present in %s, %d times\n", ch, str1, count);

    return 0;
}

// Function definition to find frequency of a character in a string
int find_frequency(char *s, char ch) {
    int count = 0;
    for(; *s; s++) {
        if (*s == ch)
            count++;
    }
    return count;
}
```

WAP to input a string and print the Square of all the Numbers in a String. Take care of special cases. Input: wha4ts12ap1p and Output: 16 1 4 1

```
#include<stdio.h>

int sqr(int r); // Function prototype for computing square of a number
void extract_num_square(char *c); // Function prototype for extracting
numbers and computing their square

int main() {
    char ch[400];
    printf("Enter a string: ");
    scanf("%s", ch); // Read a string input from the user

    extract_num_square(ch); // Call function to extract numbers and compute
their square

    return 0;
}

// Function to compute square of a number
int sqr(int r) {
    return r * r;
}

// Function to extract numbers from a string and compute their square
void extract_num_square(char *c) {
    while (*c != '\0') { // Loop until the end of the string
        if (*c >= '0' && *c <= '9') { // Check if the current character is a
digit
            printf("%c--", (*c)); // Print the digit
            printf("%d\n", sqr(*c - '0')); // Compute and print its square
        }
        c++; // Move to the next character in the string
    }
}
```

output:

```
Enter a string: prek2sh5a
2--4
5--25
```

Write a function to find the position of First occurrence of the input character in a given string.

```
#include<stdio.h>

// Function prototype for finding the position of the first occurrence of a
character in a string
int find_first(char* text, char ch);

int main() {
    char text[100];

    printf("Enter the text: ");
    scanf("%[^\n]%*c", text); // Read the text input until newline character

    printf("Enter the character you want to find: ");
    char ch = getchar();

    int res = find_first(text, ch); // Call the function to find the
position of the character

    if(res == -1)
        printf("Character not present in the text\n");
    else
        printf("Character '%c' is available at position %d\n", ch, res);

    return 0;
}

// Function to find the position of the first occurrence of a character in a
string
int find_first(char* text, char ch) {
```

```
    int position = -1; // Initialize position to -1 (indicating character
not found)

    for(int i = 0; text[i] != '\0'; i++) { // Loop through each character of
the string
        if(text[i] == ch) { // If the current character matches the desired
character
            position = i + 1; // Set the position to the current index + 1
(1-indexed position)
            break; // Exit the loop as soon as the character is found
        }
    }

    return position; // Return the position of the first occurrence of the
character
}
```

output:

```
Enter the text: preksha
Enter the character you want to find: s
Character 's' is available at position 5
```

String Matching (impppppp)

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

// Function prototype for pattern matching
int pattern_match(char text[], int n, char pat[], int m);

int main() {
    char text[100];
    char pat[100];

    printf("Enter the string: ");
    scanf(" %[^\n]%*c", text); // Read the string input until newline
```

```
character

    printf("Enter the pattern: ");
    scanf(" %[^\n]%*c", pat); // Read the pattern input until newline
character

    int n = strlen(text);
    int m = strlen(pat);

    int pos = pattern_match(text, n, pat, m);

    if(pos == -1)
        printf("Pattern not found\n");
    else
        printf("Pattern found at position %d\n", pos);

    return 0;
}

// Function to perform pattern matching
int pattern_match(char text[], int n, char pat[], int m) {
    int i, j;

    for(i = 0; i <= n - m; i++) { // Iterate through the text
        for(j = 0; j < m; j++) { // Check each character of the pattern
            if(text[i + j] != pat[j]) // If characters don't match, break
the loop
                break;
        }

        if(j == m) // If all characters of the pattern match, return the
starting position
            return i + 1; // Return 1-indexed position
    }

    return -1; // Return -1 if pattern not found
}
```

```
Enter the string: hello
Enter the pattern: ll
Pattern found at position 3
```

learn how to retrive digits and alphabets from a string

to retrive alphabets

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main() {
    char s1[100];
    scanf("%s", s1);
    char s2[100];
    int j = 0;  // index for s2
    int l = strlen(s1);
    for (int i = 0; i < l; i++) {
        if (isalpha(s1[i])) {
            s2[j] = s1[i];
            j++;
        }
    }
    s2[j] = '\0';  // null-terminate the string s2
    printf("%s", s2);

    return 0;
}
```

to retrive digits

```
#include <stdio.h>

#include <stdlib.h>
```

```
#include <ctype.h>

#include <string.h>



int main() {

char s1[100];

scanf("%s", s1);

char s2[100];

int j = 0; // index for s2

int l = strlen(s1);

for (int i = 0; i < l; i++) {

if (isdigit(s1[i])) {

s2[j] = s1[i];

j++;

}

}

s2[j] = '\0'; // null-terminate the string s2

printf("%s", s2);



return 0;

}
```

or

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main() {
    char s1[100];
    scanf("%s", s1);
    int s2[100];  // Integer array to store the digits
    int j = 0;  // index for s2

    int l = strlen(s1);
    for (int i = 0; i < l; i++) {
        if (isdigit(s1[i])) {
            s2[j] = s1[i] - '0';  // Convert character to integer
            j++;
        }
    }

    // Print the integer array
    for (int i = 0; i < j; i++) {
        printf("%d", s2[i]);
    }

    return 0;
}
```

STRUCTURE

Structure is a user-defined data type in C language which allows us to combine data of different types together.Structures provide a way of storing many different values in variables of potentially different types under the same name. Structures are generally useful whenever a lot of data needs to be grouped together.
Characteristics/properties:
• A user defined data type/non-primary/secondary
• Contains one or more components – Generally known as data members. These are named

ones.

• Collection of homogeneous or heterogeneous data members

• Size of a structure depends on implementation. Memory allocation would be at least equal to the sum of the sizes of all the data members in a structure. Offset is decided at compile time.

• The members of a structure do not occupy space in memory until they are associated with structure variable.

• Compatible structures may be assigned to each other.

Memory Allocation for Structure Variables

Coding Example_2: Below programs are run on Windows7 machine and 32 bit GCC compiler

shld be multiple of 2

```
struct test
{       int i;      char j;         };
struct test1
{       char j;       int i;        };
struct test2
{       char k;     char j;     int i;      };
struct test3
{       int i;        char k;     int j;      };
int main()
{
        printf("size of the structure is %lu\n",sizeof(struct test));    //8bytes          4+4
        struct test t;
        printf("size of the structure is %lu\n",sizeof(t));              //      8bytes      4+4
        printf("size of the structure is %lu\n",sizeof(struct test1)); //8bytes          4+4
        printf("size of the structure is %lu\n",sizeof(struct test2)); //8bytes          4+4
        printf("size of the structure is %lu\n",sizeof(struct test3)); //12bytes         4+4+4
        return 0;
```

comparison of structure

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

// Structure definition
struct testing
{
    int a;
    float b;
    char c;
};

// Function prototype
int is_equal(struct testing s1, struct testing s2);

int main()
{
    // Creating instances of the testing structure
    struct testing s1 = {44, 4.4, 'A'};
    struct testing s2 = {44, 4.4, 'A'};

    // Calling the is_equal function to compare s1 and s2
    printf("%d\n", is_equal(s1, s2)); // Output: 1 (all fields are equal)

    // Creating another instance of the testing structure
    struct testing s3 = {33, 3.3, 'A'};

    // Calling the is_equal function to compare s1 and s3
    printf("%d\n", is_equal(s1, s3)); // Output: 0 (first condition itself
fails)
    printf("%ld", (long)&s1 - (long)&s2);
    return 0;
}

// Function definition to check equality of two testing structures
int is_equal(struct testing s1, struct testing s2)
{
    // Comparing individual members of the structures
    return (s1.a == s2.a && s1.b == s2.b && s1.c == s2.c);
```

```
}
```

output:

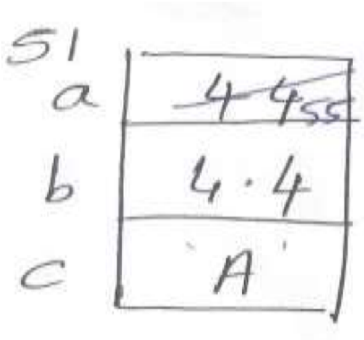
```
1
0
16
```

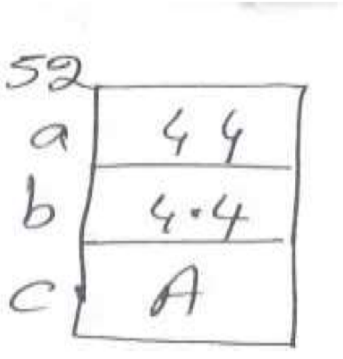
The line `printf("%d", s1 – s2);` won't compile because in C, you cannot directly subtract one structure from another using the `–` operator.

member-wise copy

Structures of same type are assignment compatible. When you assign one structure variable to another structure variable of same type, member- wise copy happens. Both of them do not point to the same memory location. The compiler generates the code for member – wise copy.

```
struct testing
{       int a;
        float b;
        char c;
};
int main()
{       struct testing s1 = {44,4.4, 'A'};
        struct testing s2 = s1;
        s1.a = 55;
        printf("%d\n",s1.a);
        printf("%d\n",s2.a);
        return 0;
}
```





1. **Structures can be assigned even if the structure contains an array. Is it True?**

Yes, structures can be assigned even if they contain arrays. When you assign one structure to another, each member of the structure, including arrays, is copied from one structure to

another.

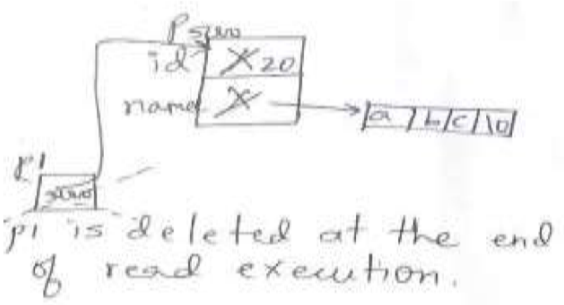
2. **If the structure contains a pointer, how member-wise copy happens?**

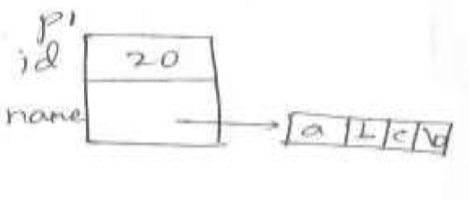
When a structure contains a pointer, and you perform a member-wise copy (like in assignment), only the pointer itself is copied, not the data it points to. This means that after the copy, both structures will have a pointer pointing to the same location in memory. If you want a deep copy, where the pointed-to data is also copied, you would need to handle that separately, usually by dynamically allocating memory for the pointer's data and then copying the data.

Functions to read and display the details of a player.

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

// Structure definition
struct player
{
    int id;
    char name[20];
};

// Function prototypes
void read(struct player *p1);
void disp(struct player p1);

int main()
{
    struct player p;
    printf("Enter id and name\n");
    read(&p); // Passing the address of p
    disp(p);
    return 0;
}

void read(struct player *p1)
{
    scanf("%d", &(p1->id)); // Using -> to access members of a pointer
    scanf(" %[^\n]", p1->name); // Correcting the format specifier for
reading the name
}
```

```
void disp(struct player p1)
{
    printf("%d\n", p1.id);
    printf("%s\n", p1.name); // Adding newline character after printing name
}
```





typedef

• typedef is a keyword which is used to give a type, a new name. By convention, uppercase letters are used for these definitions to remind the programmer that the type name is really a symbolic abbreviation, but we can use lowercase as well
• Used to assign alternative names to existing data types. It is used to provide an interface for the client. Once a new name is created using typedef, the client may use this without knowing the underlying type.
• typedef is limited to giving symbolic names to types only where as # define can be used to define values.
• typedef interpretation is performed by compiler where as # define statements are processed

by the preprocessor.

```
struct player
{       int id;
        char name[20];
};
typedef struct player player_t;     // player_t is a new name to struct player.
int main()
{
         player_t  p1;                                  // p1 is a variable
}
```

nested structure

Structure written inside another structure is called as nesting of two structures. It can be done in two ways.

1. **Separate structures:** Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct dob
{
    int date;
    int month;
    int year;
};
struct student              // template declaration which describes how student entity looks like
{
    int roll_no;            // these three are members of the structure
    char name[20];
    int marks;
    struct dob d1;                  //using structure variable of dob inside student
};
```

2. **Embedded structure:** It enables us to declare the structure inside the structure.

```
struct student              // template declaration which describes how student entity looks like
{
    int roll_no;            // these three are members of the structure
    char name[20];
    int marks;
    struct dob              //declaring structure dob inside student
    {
        int date;
        int month;
        int year;
    }d1;
};
```

Accessing Nested Structure Members: We can access the member of the nested structure by using **Outer_Structure.Nested_Structure.member** as given below:

```
int main()
{
    struct student s1 = {24, "John", 78, {20, 03, 2000}};   //Declaration and Initialization
    printf("\n Roll no. = %d, Name= %s, Marks = %d, dob = %d.%d.%d ", s1.roll_no,
    s1.name,s1.marks, s1.d1.date,s1.d1.month, s1.d1.year);
    return  0;

}
```

ARRAY OF STRUCTURE

Structure Variable Declaration:

1. **Along with the declaration of structure at the end of closing parenthesis.**

```
struct student
{
        int roll_no;
        char name[100];
        int marks;
}s[100];                    //Declaration of structure variable as s[100]
```

2. After completion of structure declaration

- Inside Main() function

```
struct student s[100];
```

- Globally using struct keyword.

```
struct student
{
        int roll_no;
        char name[100];
        int marks;
 };
struct student s[100];  // not preferred
```

In an array of structures, all elements of the array are stored in adjacent memory locations

intialization of structures

Initialization of Structures:

Compile-time Initialization: Here, user has to enter the details within the program .

.**Case 1**: struct student S1[] = { {1, "John", 60}, {2,"Jack", 40}, {3, "Jill", 77} };

// **size is decided by the compiler** based on how many students details are stored

Case 2: struct student S2[3] = { {1, "John", 60}, {2,"Jack", 40}, {3, "Jill", 77} };

// size is specified and initialized values are exactly matching with the size specified.

Case 3: struct student S3[3] = {1, "John", 60, 2,"Jack", 40, 3, "Jill", 77};

//initialization can also be done this way

Case 4 : struct student S4[5] = { {1, "John", 60} };

//**partial initialization**. Default values are stored in remaining memory locations

Runtime Initialization: Runtime is the period of time when a program is running and user is allowed to enter the input values to the variables of the structures.

code 1

```
#include <stdio.h>

struct book {
    int Number;
};

int main() {
    int n, i;
    struct book r[10];

    printf("Enter how many books you want to store: ");
    scanf("%d", &n);

    // Input book numbers
    for (i = 0; i < n; i++) {
        printf("Enter book number %d: ", i + 1);
        scanf("%d", &r[i].Number);
    }

    // Display book numbers
```

```
    printf("Book numbers:\n");
    for (i = 0; i < n; i++) {
        printf("Book %d number is: %d\n", i + 1, r[i].Number);
    }

    return 0;
}
```

output:

```
Enter how many books you want to store: 3
Enter book number 1: 123
Enter book number 2: 345
Enter book number 3: 678
Book numbers:
Book 1 number is: 123
Book 2 number is: 345
Book 3 number is: 678
```

code 2:

```
#include <stdio.h>
struct book
{
int Number;
};
int main()
{ int n,i;
struct book r[10];
printf("Enter how many books you want to store\n");
scanf("%d",&n);
printf("Enter the book number of each\n");
for( i=0;i<n;i++)
{
fflush(stdin);
scanf("%d",&r[i].Number);
}
```

```
for(i=0;i<n;i++)
{
printf("Book Number entered is:%d\n",r[i].Number); }
return 0;
}
```

pointer to array of structures

```
struct student ST[] = {{1, "John", 60}, {2, "Jack", 40}, {3, "Jill", 77}, {4, “Sam”, 78 }, {5, “Dean”, 80}};

struct student *ptr = ST;        //&ST is illogical

printf("\n%d\n", *ptr);          //prints 1

printf("%d \t %s \t %d\n", ptr->Roll_no, (ptr+1)->Name, (ptr+2)->Marks);

        //prints 1 Jack 77

ptr++;          // ptr now points to ST[1]

printf("\n%d\n", *ptr);          //prints 2

How we can use this ptr to print the details of all students??

        int i;
        for(i = 0; i < (sizeof(ST)/sizeof(ST[0])) ; i++)
        {       printf("roll_num, name and marks\n");
                printf("%d\t",(ptr+i)->roll_no);   // ptr[i].roll_no
                printf(“%s\t”, (ptr+i)->name);
                printf("%d\n",(ptr+i)->marks);
        }
```

code 1 :books problem

Let us write the C code by creating a type called Book. The Book entity contains these data members: id, title, author, price, year of publication. Write separate functions to read details of n books and display it. Also include functions to do the following.
Fetch all the details of books published in the year entered by the user.
Fetch all the details of books whose author name is entered by the user. Display appropriate

message if no data found.
Separate the interface and implementation.

Let us begin with the header file. **Header file contains the below code: book.h**

```
typedef struct Book {
    int id;
    char title[100];
    char author[100];
    int price;
    int year;
}book_t;
void read_details(book_t *b,int n); void
display_details(book_t *b, int n);
int fetch_books_year(book_t *b,int n, int year, book_t*);
int fetch_books_author(book_t *b,int n, char *author, book_t*);
```

Client code is as below: book_client.c

```
//Add appropriate header files and int main()
and return 0 with { and }
book_t b[100];
book_t b_year[100];
book_t b_author[100];
printf("How many books details you want to enter?");
int n;
scanf("%d",&n);
```

```
printf("enter the details of %d books\n",n);
read_details(b,n);
int count;
printf("enter the year of publication to find list of books in that year");
int year;
scanf("%d",&year);
count = fetch_books_year(b,n, year,b_year);
if(count)
{
      printf("List of books with %d as year of publication\n",year);
      display_details(b_year, count);
}
    else
        printf("books published in %d is not available in the dataset\n",year);
printf("\n");
printf("enter the author name to find the list of books by that author\n");
char author[100];
scanf("%s",author);
count = fetch_books_author(b,n, author,b_author); if(count)
if (count)
{
        printf("List of books by %s\n",author);
        display_details(b_author, count);
}
else
        printf("books by %s is not available in the dataset\n",author);
```

Server code is as below: book.c

```
#include "book.h"
#include "string.h"
void read_details(book_t *b,int n)
    {   int i;
        for(i = 0; i< n;i++)
        {
                printf("enter id, title, author, price and year of publication for book %d\n",i+1);
                scanf("%d%s%s%d%d",&b[i].id, b[i].title, b[i].author, &b[i].price, &b[i].year);
                // &((b+i)->id) is also valid in scanf. Using the pointer notation
        }
    }

void display_details(book_t *b, int n)
    {   int i;
        for(i = 0; i< n;i++)
        {
            printf("\n-->%d\t%s\t%s\t%d\t%d\n",b[i].id, b[i].title, b[i].author, b[i].price, b[i].year);
   // (b+i)->id is a valid pointer notation to print id
        }
    }
```



```
int fetch_books_year(book_t *b,int n, int year,book_t *b_year)
{
            int i;int count = 0; for(i = 0; i< n;i++)
            {
                if (b[i].year == year)
                {   count++;    b_year[count-1] = b[i];   }
            }
        return count;
}
```

```
int fetch_books_author(book_t *b,int n, char *author, book_t *b_author)
{
    int i;
    int count = 0;
    for(i = 0; i< n;i++)
    {
            if (!(strcmp(b[i].author, author)))
                {
                        count++;        b_author[count-1] = b[i];
                }
      }
      return count;
}
```

alternate:

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

typedef struct books {
    int id;
    char title[100];
    char author[100];
    int price;
    int year;
} book;

void read(book s[], int n);
void display(book s[], int n);
void books_year(book s[], int n, int year);
void books_author(book s[], int n, char author[]);
```

```
int main() {
    int n, year;
    char author[100];

    printf("Enter the number of books to be added: ");
    scanf("%d", &n);

    book s[n];

    read(s, n);

    printf("\nDisplaying all books:\n");
    display(s, n);

    printf("\nEnter the year to fetch books: ");
    scanf("%d", &year);
    books_year(s, n, year);

    printf("\nEnter author name to fetch books: ");
    scanf(" %[^\n]%*c", author);
    books_author(s, n, author);

    return 0;
}

// Function to read book details from the user
void read(book s[], int n) {
    for (int i = 0; i < n; i++) {
        printf("\nEnter details for book %d:\n", i + 1);
        printf("ID: ");
        scanf("%d", &s[i].id);
        printf("Title: ");
        scanf(" %[^\n]%*c", s[i].title); // To read a line of text including
spaces
        printf("Author: ");
        scanf(" %[^\n]%*c", s[i].author);
        printf("Price: ");
        scanf("%d", &s[i].price);
        printf("Year: ");
        scanf("%d", &s[i].year);
```

```
    }
}

// Function to display all book details
void display(book s[], int n) {
    for (int i = 0; i < n; i++) {
        printf("\nBook %d details:\n", i + 1);
        printf("ID: %d\n", s[i].id);
        printf("Title: %s\n", s[i].title);
        printf("Author: %s\n", s[i].author);
        printf("Price: %d\n", s[i].price);
        printf("Year: %d\n", s[i].year);
    }
}

// Function to display books published in a specific year
void books_year(book s[], int n, int year) {
    int found = 0;
    for (int i = 0; i < n; i++) {
        if (s[i].year == year) {
            found = 1;
            printf("\nBook published in year %d:\n", year);
            printf("ID: %d\n", s[i].id);
            printf("Title: %s\n", s[i].title);
            printf("Author: %s\n", s[i].author);
            printf("Price: %d\n", s[i].price);
            printf("Year: %d\n", s[i].year);
        }
    }
    if (!found) {
        printf("No books found for the year %d\n", year);
    }
}

// Function to display books by a specific author
void books_author(book s[], int n, char author[]) {
    int found = 0;
    for (int i = 0; i < n; i++) {
        if (strcmp(s[i].author, author) == 0) {
            found = 1;
```

```
            printf("\nBook by author %s:\n", author);
            printf("ID: %d\n", s[i].id);
            printf("Title: %s\n", s[i].title);
            printf("Author: %s\n", s[i].author);
            printf("Price: %d\n", s[i].price);
            printf("Year: %d\n", s[i].year);
        }
    }
    if (!found) {
        printf("No books found by the author %s\n", author);
    }
}
```

output:

```
Enter details for book 1:
ID: 1
Title: good vibes
Author: albert
Price: 345
Year: 2005

Enter details for book 2:
ID: 2
Title: calm
Author: vihu
Price: 234
Year: 1986

Enter details for book 3:
ID: 3
Title: sunrise
Author: vihu
Price: 200
Year: 2005

Displaying all books:

Book 1 details:
```

```
ID: 1
Title: good vibes
Author: albert
Price: 345
Year: 2005

Book 2 details:
ID: 2
Title: calm
Author: vihu
Price: 234
Year: 1986

Book 3 details:
ID: 3
Title: sunrise
Author: vihu
Price: 200
Year: 2005

Enter the year to fetch books: 2005

Book published in year 2005:
ID: 1
Title: good vibes
Author: albert
Price: 345
Year: 2005

Book published in year 2005:
ID: 3
Title: sunrise
Author: vihu
Price: 200
Year: 2005

Enter author name to fetch books: vihu

Book by author vihu:
ID: 2
```

```
Title: calm
Author: vihu
Price: 234
Year: 1986

Book by author vihu:
ID: 3
Title: sunrise
Author: vihu
Price: 200
Year: 2005
```

code 2: Printing the array of structures using array of pointers to structures

```
typedef struct Sample
{
    int a;
    float b;
} SAMPLE;

#include <stdio.h>


int main()
{
    SAMPLE s[] = {{2, 2.2}, {1, 1.1}, {7, 7.7}, {4, 4.4}, {3, 3.3}};
    SAMPLE* sp[1000];
    int n = sizeof(s) / sizeof(*s);

    printf("using array\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d %f\n", s[i].a, s[i].b);
    }

    printf("\nusing array of pointers\n");
    for (int i = 0; i < n; i++)
    {
```

```
        sp[i] = &s[i];
    }

    for (int i = 0; i < n; i++)
    {
        printf("%d %f\n", sp[i]->a, sp[i]->b);
    }
}
```

output:

```
using array
2 2.200000
1 1.100000
7 7.700000
4 4.400000
3 3.300000

using array of pointers
2 2.200000
1 1.100000
7 7.700000
4 4.400000
3 3.300000
```

code 3: Program to swap first and last elements of the array of structures and display the array of structures using array of structures and array of pointers

```
typedef struct Sample
{
    int a;
    float b;
} SAMPLE;

#include <stdio.h>
```

```
void disp1(SAMPLE *s, int n);  // s[]
void disp2(SAMPLE **sp, int n);  // *sp[]
void swap(SAMPLE **a, SAMPLE **b);

int main()
{
    SAMPLE s[] = {{2, 2.2}, {1, 1.1}, {7, 7.7}, {4, 4.4}, {3, 3.3}};
    SAMPLE *sp[100];
    int n = sizeof(s) / sizeof(*s);

    printf("before swap using array of structures\n");
    disp1(s, n);

    for (int i = 0; i < n; i++)
    {
        sp[i] = &s[i];
    }

    printf("before swap using array of pointers to structures\n");
    disp2(sp, n);

    // swap(&s[0], &s[n-1]);  // sp[0], sp[n-1]
    swap(&sp[0], &sp[n-1]);  // 2000, 2004

    printf("after swap using array of structures\n");
    disp1(s, n);  // no change in the original set

    printf("after swap using array of pointers to structures\n");
    disp2(sp, n);  // first and last structures are swapped.

    return 0;
}

void swap(SAMPLE **a, SAMPLE **b)
{
    SAMPLE *temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void disp1(SAMPLE *s, int n)  // SAMPLE s[]
{
    for (int i = 0; i < n; i++)
    {
        printf("%d %f\n", s[i].a, s[i].b);
    }
}

void disp2(SAMPLE **sp, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d %f\n", sp[i]->a, sp[i]->b);
    }
}
```

output:

```
before swap using array of structures
2 2.200000
1 1.100000
7 7.700000
4 4.400000
3 3.300000
before swap using array of pointers to structures
2 2.200000
1 1.100000
7 7.700000
4 4.400000
3 3.300000
after swap using array of structures
2 2.200000
1 1.100000
7 7.700000
4 4.400000
3 3.300000
after swap using array of pointers to structures
3 3.300000
1 1.100000
```

```
7 7.700000
4 4.400000
2 2.200000
```

sorting

bubble sorting

• An array is traversed from left and adjacent elements are compared and the higher one is placed at right side.
• In this way, the largest element is moved to the rightmost end at first.
• This process is continued to find the second largest number and this number is placed in the second place from rightmost end and so on until the data is sorted.

code 1:Using array of structures, sort the structures in the array based on the data member – roll_no

```
#include <stdio.h>

typedef struct Student {
    int marks;
    char name[100];
    int roll_no;
    float cgpa;
} STUDENT;

void input_data(STUDENT *s, int n);
void display_data(STUDENT *s, int n);
void sort(STUDENT *s, int n);

int main() {
    STUDENT s[1000];
    int n;
    printf("Enter the number of students: ");
    scanf("%d", &n);

    input_data(s, n);
```

```
    printf("Entered details before sorting:\n");
    display_data(s, n);

    sort(s, n);

    printf("After sorting, the sorted list printed using the array of
structures:\n\n");
    display_data(s, n);

    return 0;
}

void input_data(STUDENT *s, int n) {
    for (int i = 0; i < n; i++, s++) {
        printf("Enter the details as marks, name, roll_no, cgpa: \n");
        scanf("%d %s %d %f", &(s->marks), s->name, &(s->roll_no), &(s-
>cgpa));
    }
}

void sort(STUDENT *s, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n - i; j++) {
            if (s[j - 1].roll_no > s[j].roll_no) {
                STUDENT t = s[j - 1];
                s[j - 1] = s[j];
                s[j] = t;
            }
        }
    }
}

void display_data(STUDENT *s, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d\t%s\t%d\t%.2f\n", s[i].marks, s[i].name, s[i].roll_no,
s[i].cgpa);
    }
}
```

output:

```
Enter the number of students: 2
Enter the details as marks, name, roll_no, cgpa:
89
nidhi
78
9.3
Enter the details as marks, name, roll_no, cgpa:
90
neda
89
9.5
Entered details before sorting:
89      nidhi   78      9.30
90      neda    89      9.50
After sorting, the sorted list printed using the array of structures:

89      nidhi   78      9.30
90      neda    89      9.50
```

code 2:

Coding Example_3: Using array of pointers to structures, sort the structures in the array based on the data member – roll_no , without affecting the array of structures

```
#include<stdio.h>
typedef struct Student
{
        int marks;
        char name[100];
        int roll_no;
        float cgpa;
} STUDENT;

void input_data(STUDENT *s, int n);
void sort(STUDENT **, int n);
void init_structures_pointers(struct Student **sp,struct Student *s,int n);
void display_data_pointers(STUDENT **sp,int n);
void display_data(STUDENT *s, int n);
```

```
int main()
{
        STUDENT s[1000];
        int n;
        printf("Enter the number of students: ");
        scanf("%d", &n);
        input_data(s,n);
```

```
        struct Student* sp[n];
        init_structures_pointers(sp,s,n);
        sort(sp,n);
        printf("After sorting,printing the list using array of structures\n");
        display_data(s,n); // Observe that there is no change in the array of structures.
        printf("After sorting,printing the list using array of pointers\n");
        display_data_pointers(sp,n);  // observe the sorted list printed using pointers



        return 0;
}
void init_structures_pointers(struct Student **sp,struct Student *s,int n)
{
        int i;
        for( i = 0 ; i < n; i++)
        {
                sp[i] = &s[i]; //(s+i)
        }
}
```

```
void sort(struct Student *sp[],int n)
{
        int i,j;
        for( i = 0 ; i < n ; i++)
        {

                for( j = 1; j < n-i; j++)
                {
                        if(sp[j-1]->roll_no > sp[j]->roll_no)
                        {

                        STUDENT *t;
                        t = sp[j-1];
                        sp[j-1] = sp[j];
                        sp[j] = t;
                    }
                }
        }
}
void input_data(STUDENT *s, int n)
{
        int i;

        for(  i = 0 ; i < n; i++)
        {
                printf("Enter the details as marks,name,roll_no,cgpa \n");
                scanf("%d%s%d%f", &((s+i)->marks), (s+i)->name, &(s[i].roll_no), &(s[i].cgpa));

        }
}
```

```
void display_data_pointers(STUDENT **sp,int n)
{
        int i;
        for( i = 0; i < n ; i++)
        {
                printf("%d\t%s\t%d\t%0.2f\n", sp[i]->marks, sp[i]->name,(*(sp[i])).roll_no, sp[i]->cgpa);
        }
}

void display_data(STUDENT *s, int n)
{
        int i;
        for( i = 0 ; i < n; i++)
        {

                printf("%d\t%s\t%d\t%0.2f\n", (s[i].marks), s[i].name, (s[i].roll_no), (s[i].cgpa));
        }
}
```

LINK LIST

Introduction

Linked List can be defined as collection **of objects called nodes that are randomly stored and logically connected in the memory via links.** The node and link has a special meaning which we will be discussing in this chapter. Before we deal with this in detail, we need to answer few questions.

basics

Can we have a pointer data member inside a structure? Yes. Refer to below codes. (refer all the versions)

```
#include <stdio.h>

struct A {
```

```
    int a;
    int *b;
};

int main() {
    struct A a1;
    struct A a2;

    a1.a = 100;
    int c = 200;
    a1.b = &c;

    printf("a1 values: %d and %d\n", a1.a, *(a1.b));
    a2 = a1;
    printf("a2 values: %d and %d\n", a2.a, *(a2.b));
    // Output: 100 200
    // Output: 100 200

    a1.a = 300;
    *(a1.b) = 400;

    printf("a2 values: %d and %d\n", a2.a, *(a2.b));
    // Output: 100 400

    printf("a1 values: %d and %d\n", a1.a, *(a1.b));
    // Output: 300 400

    struct A *p = &a1;
    printf("a1 values: %d and %d\n", p->a, *(p->b));
    // Output: 300 400

    return 0;
}
```

```
a1 values: 100 and 200
a2 values: 100 and 200
a2 values: 100 and 400
```

```
a1 values: 300 and 400
a1 values: 300 and 400
```

code 2:

```
struct Sample {

int a;

int *b;

};

#include<stdio.h>

int main() {

struct Sample s;

s.a = 100;

s.b = &(s.a);
printf("%d %d",s.a,*(s.b));
struct Sample s1;

s1.a = 100;

s1.b = &(s1.a);
printf("%d %d\n",s1.a,*(s1.b));
struct Sample s2 = s1;
printf("%p %p\n",s1.b,s2.b);
printf("%d %d\n",s2.a,*(s2.b));
s2.a = 200;

printf("%p %p\n",s1.b,s2.b);
printf("%d %d\n",s1.a,*(s1.b));

printf("%d %d\n",s2.a,*(s2.b)); // very imp

*(s2.b) = 300;
```

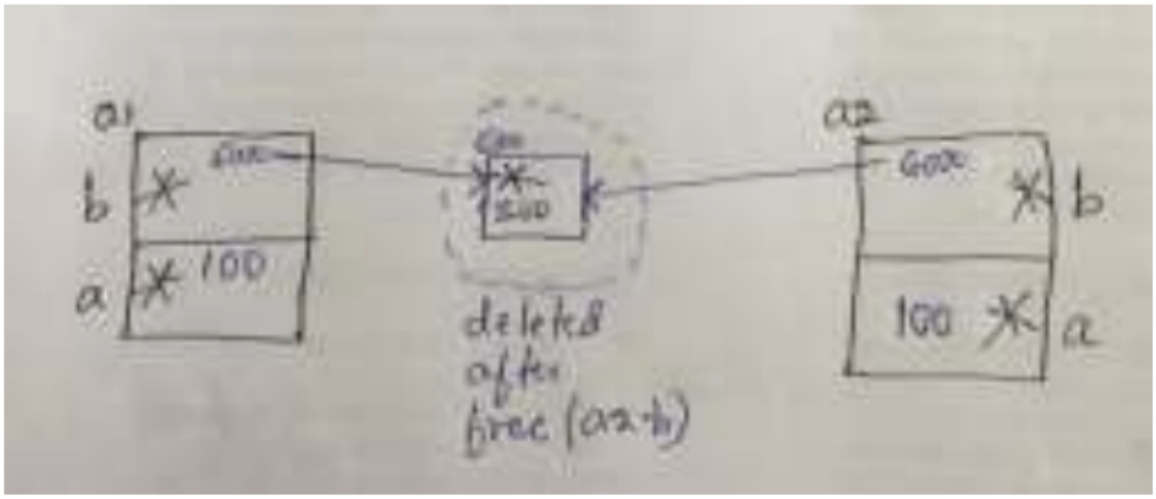
```
printf("%p %p\n",s1.b,s2.b);

printf("%d %d\n",s1.a,*(s1.b));

printf("%d %d\n",s2.a,*(s2.b));

s2.b = &(s2.a);

*(s2.b) = 400;

printf("%p %p\n",s1.b,s2.b);

printf("%d %d\n",s1.a,*(s1.b));

printf("%d %d\n",s2.a,*(s2.b));

return 0;

}
```

output:

```
100 100100 100
0x16f7db268 0x16f7db268
100 100
0x16f7db268 0x16f7db268
100 100
200 100
0x16f7db268 0x16f7db268
300 300
200 300
0x16f7db268 0x16f7db250
300 300
400 400
```

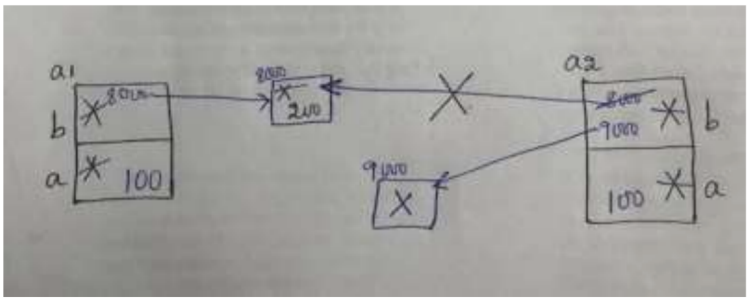
Version 3:

```
int main()
{
        struct A a1; struct A a2; a1.a = 100;
        a1.b = (int*) malloc(sizeof(int));
        *(a1.b) = 200;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        a2 = a1;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 200
        free(a2.b); // a1.b too becomes dangling pointer
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 undefined behaviour
        return 0;
}
```



imp

```
return 0;
}
```



Version 4:

```
int main() {
        struct A a1; struct A a2; a1.a = 100;
        a1.b = (int*) malloc(sizeof(int));
        *(a1.b) = 200;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        a2 = a1;
        a2.b = (int*) malloc(sizeof(int));
        // changing this to a1.b creates garbage and output differs
        printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 junk value
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        free(a1.b);     free(a2.b);     return 0;
}
```

Can we have a structure variable inside another structure? Yes. Refer to below code.

```
#include<stdio.h>
struct A
{
int a;
};
struct B {
int a;
struct A a1;
}; // structure variable a1 as a data member in B
int main() {
printf("sizeof A %d\n",sizeof(struct A));
printf("sizeof B %d\n",sizeof(struct B)); // more than A
return 0;
}
```

output:

```
4
8
```

Q3. Can we have a structure variable inside the same structure? – No.

Coding Example_3:

```
struct A {
        int a;
        struct A a1;
};
```

//Compile time Error: field a1 has incomplete type

// size of struct A we cannot find as the field a1 which itself is a structure of the same kind.

The solution to the previous version is to **have a pointer of the same type inside the structure**. The size of a pointer to any type is fixed. Hence, the size of the structure can be computed by the compiler at compile time.

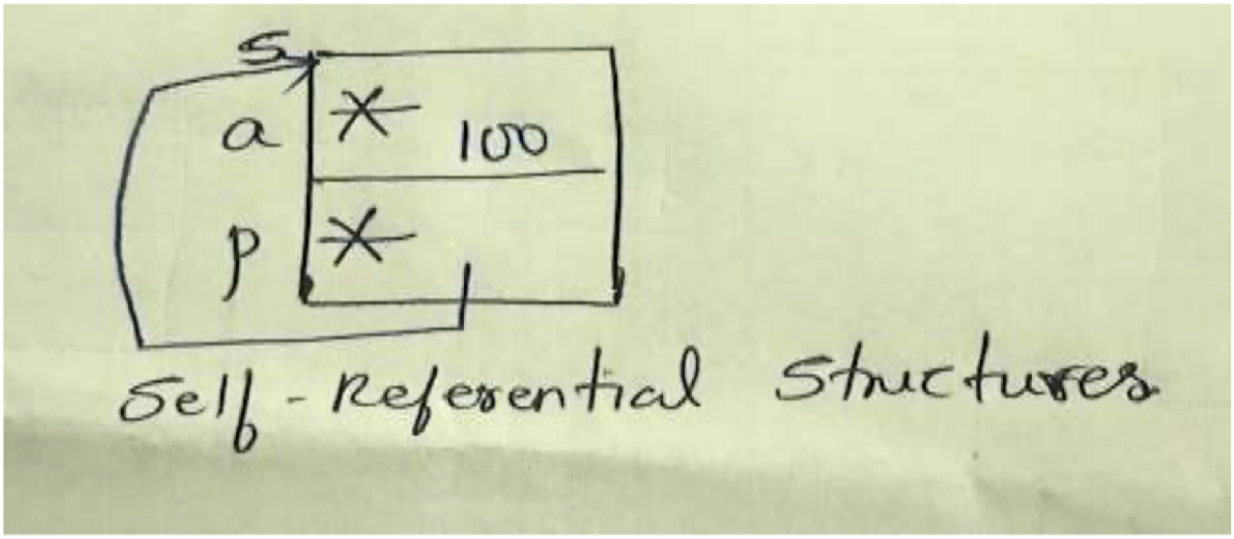
Self Referential Structures

A structure which has a **pointer to itself as a data member** is called a self - referential structure. In this type of structure, the object of the same structure points to the same data structure and refers to the data types of the same structure. It can have one or more pointers pointing to the same type of structure as their member. It is **widely used in dynamic data structures such as trees, linked lists**, etc.

imppp

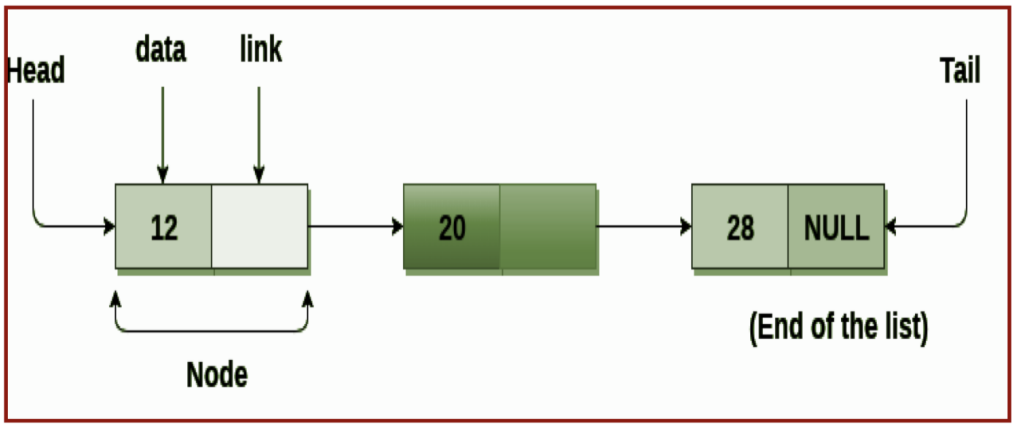
Coding Example_ 4:

```
#include<stdio.h>
struct Sample
{
        int a;
        struct Sample *p;
};
int main()
{
        printf("%d\n",sizeof(struct Sample) ); // implementation specific
        struct Sample s;
        s.a = 100;
        s.p = &s;
        printf("%d %d %d\n", s.a, s.p->a, s.p->p->a); // all 100
        return 0;
}
```



Linked List

Linked List can be defined as collection of objects called **nodes that are randomly stored and logically connected in the memory via links**. A node **contains minimum two fields** - data stored at a particular address and the pointer which contains the address of the next node in the memory. The **last node of the list contains pointer to the null.** The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list, which is used for optimized utilization of space. **Size of the list is limited to the memory size and doesn't need to be declared in advance.**



Characteristics

1. A data structure that consists of zero or more nodes. Every node is composed of minimum two fields: a data/component field and a pointer field. The pointer field of every node point to the next node in the sequence.
2. We can access the nodes one after the other. There is no way to access the node directly as random access is not possible in a linked list. Lists have sequential access.
3. Insertion and deletion in a list at a given position requires no shifting of elements.

Differences between Arrays and Linked List

1. Array - a collection of same type data elements

   Linked list -a collection of unordered linked elements known as nodes.

2. Array- traversal through indexes.

   Linked list - traversal through the head until we reach the last node.

3. Array - Elements are stored in contiguous address space

   Linked List - Elements are at random address spaces.

---

4. Array – Random access is faster.

   Linked List – Random access is slower.

5. Array- Insertion, and Deletion of an element is not that efficient.

   Linked List - Insertion and Deletion of an element is efficient.

6. Array - Fixed Size.

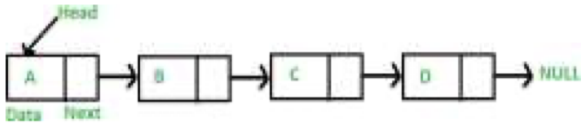
   Linked List – Dynamic Size

7. Array - Memory allocation decided during compile time/ static allocation.

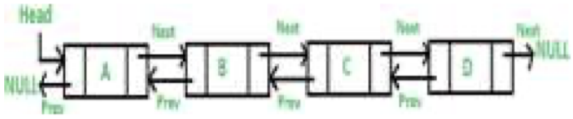
   Linked List - Memory allocation decided during runtime / dynamic allocation.

Types of Linked List

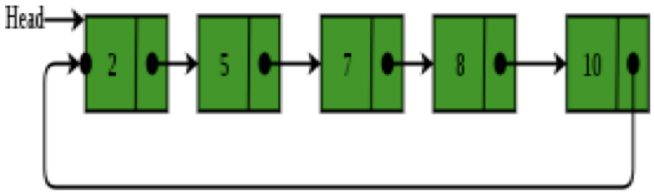
i. Singly Linked List



ii. Doubly Linked List



iii. Circular Linked List



Operations on Linked List

- **Insertion Operation on the list** includes Insertion at the beginning of the list, Insertion at the end of the list and Insertion at a Specific Node in the List
- **Deletion Operation on the list** includes Deletion at the beginning of the list, Deletion at the end of the list and Deletion of a Specific Node in the List
- **Other Operations on the list** such as Traversing the list, Searching the list and Sorting the List, etc,

imppp code(learn)

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

typedef struct node {
    int info;
    struct node *next;
```

```
} NODE;

NODE* insertFront(NODE* head, int ele);
void display(NODE *head);
NODE* deleteFront(NODE* head, int *pele);
NODE* freeList(NODE* head);

int main() {
    NODE *head = NULL;
    int choice;
    int ele;

    do {
        printf("1. InsertFront 2. Display 3. DeleteFront 4. Exit\n");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter an integer\n");
                scanf("%d", &ele);
                head = insertFront(head, ele);
                break;
            case 2:
                display(head);
                break;
            case 3:
                if (head != NULL) {
                    head = deleteFront(head, &ele);
                    printf("Deleted element is %d\n", ele);
                } else {
                    printf("List is already empty\n");
                }
                break;
            case 4:
                head = freeList(head);
                break;
            default:
                printf("Invalid choice. Please enter a number between 1 and
4.\n");
        }
    } while (choice != 4);
```

```
    return 0;
}

NODE* createNode(int ele) {
    NODE* newNode = malloc(sizeof(struct node));
    // We assume memory is always allocated to the newNode and
    // hence not checking for newNode == NULL
    newNode->info = ele;
    newNode->next = NULL;
    return newNode;
}

NODE* insertFront(NODE* head, int ele) {
    NODE* newNode = createNode(ele);
    newNode->next = head;
    head = newNode;
    return head;
}

void display(NODE *head) {
    if (head == NULL) {
        printf("Empty List\n");
    } else {
        NODE *p = head;
        while (p != NULL) {
            printf("%d ", p->info);
            p = p->next;
        }
        printf("\n");
    }
}

NODE* deleteFront(NODE* head, int *pele) {
    if (head == NULL) {
        return NULL;
    }
    NODE *p = head;
    *pele = head->info;
    head = head->next;
```

```
    free(p);
    return head;
}

NODE* freeList(NODE* head) {
    NODE *p = head;
    while (head != NULL) {
        head = head->next;
        printf("Freeing %d\n", p->info);
        free(p);
        p = head;
    }
    return NULL;
}
```

link list extended

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

typedef struct node {
    int info;
    struct node *next;
} NODE;

NODE *insertFront(NODE *head, int ele);
void display(NODE *head);
NODE *insertpos(NODE *head, int ele1, int pos);
NODE *insertRear(NODE *head, int ele);

int main() {
    NODE *head = NULL;
    int choice;
    int ele;
    int ele1;
    int pos;

    do {
        printf("1. InsertFront 2. Display 3. InsertPosition 4. InsertRear 5.
```

```
Exit\n");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf("Enter an integer:\n");
            scanf("%d", &ele);
            head = insertFront(head, ele);
            break;
        case 2:
            display(head);
            break;
        case 3:
            printf("Enter an integer:\n");
            scanf("%d", &ele1);
            printf("Enter the position where the node needs to be
inserted:\n");
            scanf("%d", &pos);
            head = insertpos(head, ele1, pos);
            break;
        case 4:
            printf("Enter the number to be inserted at the end:\n");
            scanf("%d", &ele);
            head = insertRear(head, ele);
            break;  // Add break here
        case 5:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice. Please enter a number between 1 and
5.\n");
        }
    } while (choice != 5);

    return 0;
}

NODE *createNode(int ele) {
    NODE *newNode = malloc(sizeof(struct node));
    newNode->info = ele;
    newNode->next = NULL;
```

```
    return newNode;
}

NODE *insertFront(NODE *head, int ele) {
    NODE *newNode = createNode(ele);
    newNode->next = head;
    head = newNode;
    return head;
}

void display(NODE *head) {
    if (head == NULL) {
        printf("Empty List\n");
    } else {
        NODE *p = head;
        while (p != NULL) {
            printf("%d ", p->info);
            p = p->next;
        }
        printf("\n");
    }
}

NODE *insertpos(NODE *head, int ele1, int pos) {
    NODE *newNode = createNode(ele1);
    if (pos == 0) {
        newNode->next = head;
        return newNode;
    }

    NODE *temp = head;
    for (int i = 0; temp != NULL && i < pos - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds\n");
        free(newNode);
        return head;
    }
```

```
    newNode->next = temp->next;
    temp->next = newNode;

    return head;
}

NODE *insertRear(NODE *head, int ele) {
    NODE *newNode = createNode(ele);

    if (head == NULL) {  // If the list is empty, new node becomes the head
        return newNode;
    }

    NODE *tail = head;

    // Traverse to the end of the list
    while (tail->next != NULL) {
        tail = tail->next;
    }

    // Insert the new node at the end
    tail->next = newNode;

    return head;
}
```

UNION

A Union is a user-defined datatype in the C programming language. It is a collection of variables of different datatypes in the same memory location. We can define a union with many members, but at a given point of time, only one member can contain a value. Union unlike structures, share the same memory location. Using Union in C will save Memory Space in a given context.

C unions are used to save memory. To better understand a union, think of it as a chunk of memory that is used to store variables of different types. When we want to assign a new value to a field, the existing data is replaced with new data. C **unions allow data members which are mutually exclusive to share the same memory**. This is quite important when memory is valuable, such as in embedded systems. Unions are mostly used in embedded programming where direct access to memory is needed.

You can define a union with many members, but **only one member can contain a value at any given time. The memory occupied by a union will be large enough to hold the largest member of the union. So, the size of a union is at least the size of the biggest component. At a given point in time, only one can exist. All the fields overlap and they have the same offset: 0.**

Coding Example_1: Displays the total memory size occupied by the union

```
#include <stdio.h>
union car
{
        char name[10];          //assuming, 1 byte for char
        float price;            // assuming, 4 bytes for float
};
int main( )
{
        union car c;
        printf( "Memory size occupied by data in bytes : %d\n", sizeof(union car));
        return 0;
}
```

Output: Memory size occupied by data in bytes : 12

Note: Output may vary depending on the number of bytes of padding

Coding Example_2:

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

union car
{
      char name[10];
      float price;
};
```

```
int main()
{
        union car c;
        strcpy(c.name, "Benz");
        c.price=4000000.00;
        printf( "car name: %s\n", c.name);
        printf( "car price: %f\n", c.price);
        return 0;
}
```

Output: When the above code is compiled and executed, the value of name gets corrupted because the final value assigned to the variable price has occupied the memory location, and this is the reason that the value of price member is printed well.

```
#include <stdio.h>
#include<string.h>
union car
{
        char name[10];
        float price;
};
int main( )
{       union car c;
        strcpy( c.name, "Benz");
        printf( "car name: %s\n", c.name);
        c.price=4000000.00;
        printf( "car price: %f\n",c.price);
        return 0;
  }
```

Output:

```
        Car name: Benz
        Car price:4000000.000000
```

Coding Example_4: Size of union and accessing the union members

```
#include<stdio.h>

union X
{       int i;
        int j;
        double k;
};              // ; compulsory

struct Y
{       int i;
        int j;
        double k;
};
int main()
{
        printf("sizeof X is %lu\n",sizeof(union X));
        union X x1;
        x1.i = 23;
        x1.j = 23;
        x1.i = 49;
        printf("%d %d\n",x1.i, x1.j);
        printf("sizeof Y is %lu\n",sizeof(struct Y));
        return 0;
}
```

Output:

```
sizeof X is 8
49 49
sizeof Y is 16
```

Coding Example_5: Size of union and accessing the union members

```
#include<stdio.h>
union Z
{
        int  a;
        int b[3];
};
int main()
{
        union Z z;
        z.a = 23;
        printf("size of Z is %d bytes\n",sizeof(z));
        printf("%d %d %d\n",z.b[0], z.b[1],z.b[2]);
        return 0;
}
```

Output:

size of Z is 12 bytes

23 0 2179072 // 23 junk value junk value

Coding Example_6: All the fields overlap and they have the same offset: 0 in union

```
#include<stdio.h>
#include<stddef.h> // offsetof function is declared in stddef.h
union A
{
        int x;
        int y;
        int z;
};
struct B
{
        int x;
```

```
        int y;
        int z;
};
int main()
{
        // assumption int occupies four bytes
        printf("%lu\n",offsetof(union A,y)); // 0
        printf("%lu\n",offsetof(struct B,y));  // 4
        printf("%lu\n",offsetof(struct B,z));  // 8
}
```

offset-is the position where it is stored

impp

Structure:

1. The keyword struct is used to define a structure
2. When a variable is associated with a structure, memory is allocated to each member of the structure. The size of the structure is greater than or equal to the sum of its members.
3. Each member within a structure is assigned unique storage area location.
4. In Structure, the address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.
5. In Structure, altering the value of a member will not affect other members of the structure.
6. All members can be accessed at a time.

Union:

1. The keyword union is used to define a union.
2. When a variable is associated with a union, memory is allocated by considering the size of the largest data member. So, the size of a union is at least equal to the size of its largest member.
3. Memory allocated for union is shared by individual members of union
4. For unions, the address is same for all the members of a union. This indicates that every member begins at the same offset.
5. Altering the value of any of the member will alter other member values.
6. In Unions, only one member can be accessed at a time.

Similarites between structures and unions:

1. Both are user-defined data types used to store data of same or different types as a single unit.
2. Their members can be objects of any type, including other structures, unions, or arrays. A member can also consist of a bit of a field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must be of the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. **". "**operator is used for accessing members.

code 1:

Coding Example_7: Structure containing the union

```
#include<stdio.h>
struct test
{       int i;
        union test2
        {
                char a[20];
                float k;
                int j;
        }u;
};
int main()
{
        printf("%lu\n",sizeof(struct test));      // 24
        union test2 t2;             // allowed
```

```
        t2.k=45.6;
        printf("k is %f\n",t2.k);
        struct test t;
        t.u.j=78;
        printf("j is %d\n",t.u.j);
        t.u.k=45.6;
        printf("k is %f\n",t.u.k);
        return 0;
}
```

Output:

24

k is 45.599998

j is 78

k is 45.599998

code 2:

Coding Example_8: Usage of anonymous union inside a structure

```
#include<stdio.h>
struct student
{
        union
        {                                   //anonymous union (unnamed union)
                char name[10];
                int roll;
        };
         int mark;
 };
 int main()
  {
        struct student stud;
        char choice;
        printf("\n You can enter your name or roll number ");
```

```
        printf("\n Do you want to enter the name (y or n): ");
        scanf("%c",&choice);
        if(choice=='y'||choice=='Y')
        {       printf("\n Enter name: ");
                scanf("%s",stud.name);
                printf("\n Name:%s",stud.name);
        }
        else
        {       printf("\n Enter roll number");
                scanf("%d",&stud.roll);
                printf("\n Roll:%d",stud.roll);
        }
        printf("\n Enter marks");
        scanf("%d",&stud.mark);
        printf("\n Marks:%d",stud.mark);
        return 0;
}
```

Output:

You can enter your name or roll number

Do you want to enter the name (y or n) : y

Enter name: john

Name:john

Enter marks: 45

Marks:45

Coding Example_9: Usage of anonymous union inside a union

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
union test
```

```
{
        int i;
        union
        {
                char a[20];
                float k;
        };
};
int main()
{
        printf("%lu",sizeof(union test));        // 20
        union test t;
        t.i=78;                        // One member at a time from union
        printf("\ni is %d\n",t.i);
        strcpy(t.a,"sindhu");
        printf("a is %s\n",t.a);
        t.k=45.5;
        printf("k is %f",t.k);
        return 0;
}
```

Output:

20

i is 78

a is sindhu

k is 45.500000

Coding Example_10: Structure inside Union

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



```
{
        char name[30];
        int rollno;
        float percentage;
};
union details
{
        struct student s1;
};
```

```
int main()
{
        union details  set;
        printf("Enter details:");
        printf("\nEnter name : ");
        scanf("%s", set.s1.name);
        printf("\nEnter roll no : ");
        scanf("%d", &set.s1.rollno);
        printf("\nEnter percentage :");
        scanf("%f", &set.s1.percentage);
        printf("\nThe student details are : \n");
        printf("\nName : %s", set.s1.name);
        printf("\nRollno : %d", set.s1.rollno);
        printf("\nPercentage : %f", set.s1.percentage);
        return 0;
}
```

Output:

```
Enter details:
Enter name : sindhu
Enter roll no : 123
Enter percentage :67
```

The student details are :

Name : sindhu

Rollno : 123

Percentage : 67.000000

bitfields

In C, we can specify the size (in bits) of the **structure and union members**. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range. The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit.

Declaration:

```
struct
{
        type [member_name] : width ;
};
```

type - An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.

member_name -The name of the bit-field.

width -The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Coding Example_1:

```
#include<stdio.h>
struct Status
{
        unsigned int bin1:1; // 1 bit is allocated for bin1. only two digits can be stored 0 &1.
```

```
        unsigned int bin2:1;
          // if it is signed int bin1:1   or int bin1:1,        one bit is used to represent the sign
};
int main()
{
         printf("Size of structure is %lu\n",sizeof(struct Status));      // 4 bytes
         struct Status s;
         printf("enter the number");
        //scanf("%d",&s.bin1);    // Error
         // We cannot access the address of bit - field. system is byte addressable.
         return 0;
 }
```

Coding Example_2: Demonstration of the maximum value that can be stored in bitfields. Results in warning. If you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as below. The age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so.

```
struct {
        unsigned int age : 3;
} Age;
#include <stdio.h>
#include <string.h>
struct
{
        unsigned int age : 3;
} Age;
int main( )
{
          Age.age = 4;
          printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

            printf( "Age.age : %d\n", Age.age );
            Age.age = 7;
            printf( "Age.age : %d\n", Age.age );
            Age.age = 8;
            printf( "Age.age : %d\n", Age.age );
            return 0;
    }
```

When the above code is compiled it will compile with a warning as shown

```
Coding Exampe_2.c: In function 'main':
17:14: warning: unsigned conversion from 'int' to 'unsigned char:3' changes value from '8' to '0' [-
Woverflow]
   Age.age = 8;
             ^
```

Output:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Coding Example_3: Demonstration of assigning signed value to an Unsigned variable for which bit fields are specified.

```
#include<stdio.h>
struct Status
{
        unsigned int bin1:4;   // 4 bits is allocated for bin1. 0 to 15, any number can be used.
        unsigned int bin2:2;   // 2 bits allocated for bin2. 0 to 3, any number can be used
};
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));          // again 4 bytes
        struct Status s1;

          s1.bin1=-7;     // it is unsigned. but sign is given while assigning
          s1.bin2=2;
          printf("%d %d",s1.bin1,s1.bin2);   // 9 2
          return 0;
  }
```

Coding Example_4: Structure having two members with signed and unsigned variable for which bit fields are specified.

```
#include<stdio.h>
 struct Status
{
            int bin1:4; // one bit is used for representing the sign. Another 3 bits for data
            unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used
};
int main()
{
            printf("Size of structure is %lu\n",sizeof(struct Status));  // again 4 bytes
            struct Status s1;
            s1.bin1=-7; s1.bin2=2;
            printf("%d %d",s1.bin1,s1.bin2);                                    // -7 2
            return 0;
}
```

Coding Example_5: Array of bit fields not allowed. Below code results in Error.

```
#include<stdio.h>

struct Status
{
        unsigned char bin1[10]:40;     // invalid type
};
```



```
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));
        return 0;
}
```


Coding Example_6: We cannot have pointers to bit field members as they may not start at a byte boundary. This code results in error.

```
#include<stdio.h> struct
Status
{
        int bin1:4; // 1 bit is used for representing the sign. Another 3 bits for data
        int *p:2;
};
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));
        return 0;
}
```

Coding Example_7: Storage class can't be used for bit field. This code results in error.

```
#include<stdio.h>
struct Status
{
        static int bin1:32;// Any storage class not allowed for bit field
        int p:2;
};
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));
        return 0;
}
```

Coding Example_8: It is implementation defined to assign an out-of-range value to a bit field member. This code results in error.

```
#include<stdio.h>
struct Status
{
        int bin1:33;    // Error: width exceeds its type
        int p:2;
};
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));
        return 0;
}
```

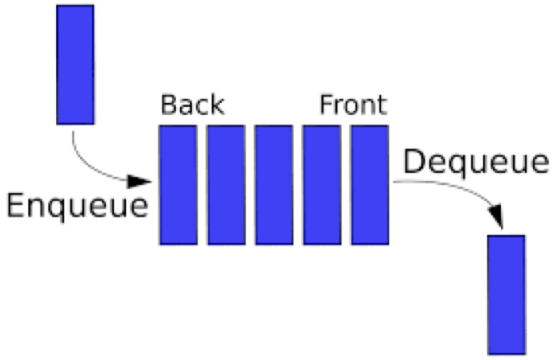
Coding Example_ 9: Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field can cause the next field to be aligned on the next container boundary where the container is the same size as the underlying type of the bit field. Below code demonstrates the force alignment on next boundary.

```
#include<stdio.h>
struct Status
 {
        int bin1:4;      // one bit is used for representing the sign. Another 3 bits for data
        //int i:0;// you cannot allocate 0 bits for any member inside the structure. Error
         int :0;// A special unnamed bit field of size 0 is used to force alignment on
        // next boundary. observe no member variable. only size is 0 bits
          unsigned int bin2:2;    // 2 bits allocated for bin2. 0 to 3, any number can be used
 };
 int main()
 {
        // observe the size of the structure. Now 8 bytes

          printf("Size of structure is %lu\n",sizeof(struct Status));          // 8 bytes
          struct Status s1;
          s1.bin1=-7;
          s1.bin2=2;
          printf("%d %d",s1.bin1,s1.bin2);          // -7 2
          return 0;
 }
```

queue

Introduction

Queue: A line or a sequence of people or vehicles waiting for their turn to be attended or to proceed. In computer Science, **a list of data items, commands, etc., stored so as to be retrievable in a definite order. A Linear data structure which has 2 ends - Rear end and a Front end**. Data elements are inserted into the queue from the rear (back) end and deleted from the front end. Hence a queue is also known as **First In First Out (FIFO)** data structure.



Two major Operations on a queue:

- enqueue() – add (store) an item to the queue from rear end.
- dequeue() – remove (access) an item from the queue from front end.

Others operations may include,

- peek() – Gets the element at the front of the queue without removing it.
- isfull() – Checks if the queue is full.
- isempty() – Checks if the queue is empty.

Types of Queue:

- **Ordinary queue** - insertion takes place at rear and deletion takes place at front.
- **Priority queue** - special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue
- **Circular queue** - Last element points to first element of queue making circular link.
- **Double ended queue** - insertion and removal of elements can be performed from either from the front or rear.

Priority Queue

A data structure which is like a regular **queue** but where additionally each element has a "**priority**" associated with it. This priority decides about the deque operation. The enqueue operation stores the item and the priority information.

Types of Priority Queue:

- **Ascending priority queue:**
  - The smallest number, the highest priority.
- **Descending priority queue:**
  - The highest number, the highest priority.

Applications of Priority Queue:

1. Dijkstra's Algorithm
2. Prims Algorithm
3. Data Compression
4. Artificial Intelligence
5. Heap Sort

Priority Queue can be implemented using different ways.

- Using an Unordered Array
- Using an Ordered Array
- Using an Unordered Linked list
- Using an Ordered Linked list
- Using Heap