POINTERS

A pointer is a variable which contains the address in memory of another variable.

The two most important operator used with the pointer type are

& - The unary operator ***&*** which gives the address of a variable

* - The indirection or dereference operator * gives the content of the object pointed to by a pointer.

Declaration

```
int i, *pi;
```

Here, ***i*** is the integer variable and ***pi*** is a pointer to an integer

```
pi = &i;
```

Here, &i returns the address of ***i*** and assigns it as the value of ***pi***

Null Pointer

The null pointer points to no object or function.
The null pointer is represented by the integer 0.
The null pointer can be used in relational expression, where it is interpreted as false.

Ex: if (pi = = NULL) or if (!pi)

Pointers can be Dangerous:

Pointer can be very dangerous if they are misused. The pointers are dangerous in following situations:

1. Pointer can be dangerous when an attempt is made to access an area of memory that is either out of range of program or that does not contain a pointer reference to a legitimate object.

Ex:
```
main ()
    {
        int *p;
        int pa = 10;
        p = &pa;
        printf("%d", *p);     //output = 10;
        printf("%d", *(p+1)); //accessing memory which is  out of range
    }
```

2. It is dangerous when a NULL pointer is de-referenced, because on some computer it may return 0 and permitting execution to continue, or it may return the result stored in location zero, so it may produce a serious error.

3. Pointer is dangerous when use of explicit **type casts** in converting between pointer types

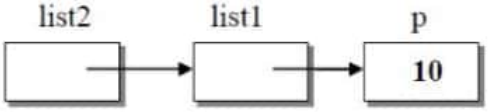
Ex:
```
pi = malloc (sizeof (int));
pf = (float*) pi;
```

4. In some system, pointers have the same size as type **int**, since **int** is the default type specifier, some programmers omit the return type when defining a **function**. The return type defaults to **int** which can later be interpreted as a pointer. This has proven to be a dangerous practice on some computer and the programmer is made to define explicit types for functions.

Pointers to Pointers

A variable which contains address of a pointer variable is called pointer-to-pointer.

Example:
```
int p;
int *list1, * *list2;
p=10;
list1=&p;
list2=&list1;
printf("%d, %d, %d", a, *list1, **list2);
```



Output: **10 10 10**

DYNAMIC MEMORY ALLOCATION FUNCTIONS

1. malloc( ):

The function *malloc* allocates a user- specified amount of memory and a pointer to the start of the allocated memory is returned.

If there is insufficient memory to make the allocation, the returned value is NULL.

Syntax:

```
data_type *x;
x= (data_type *) malloc(size);
```

Where,

x is a pointer variable of data_type

size is the number of bytes

Ex:

```
int *ptr;
ptr = (int *) malloc(100*sizeof(int));
```

2. calloc( ):

The function *calloc* allocates a user- specified amount of memory and initializes the allocated memory to **0** and a pointer to the start of the allocated memory is returned.

If there is insufficient memory to make the allocation, the returned value is NULL.

Syntax:

```
data_type *x;
x= (data_type *) calloc(n, size);
```

Where,

x is a pointer variable of type int

n is the number of block to be allocated

size is the number of bytes in each block

Ex:

```
int *x
x= calloc (10, sizeof(int));
```

The above example is used to define a one-dimensional array of integers. The capacity of this array is n=10 and x [0: n-1] (x [0, 9]) are initially 0

Macro CALLOC

```
#define CALLOC  (p, n, s)\
if ( ! ((p) = calloc (n, s)))\
    {\
        fprintf(stderr, "Insuffiient memory");\
        exit(EXIT_FAILURE);\
    }\
```

3. realloc( ):

- Before using the realloc( ) function, the memory should have been allocated using malloc( ) or calloc( ) functions.
- The function relloc( ) resizes memory previously allocated by either ***mallor*** or ***calloc***, which means, the size of the memory changes by extending or deleting the allocated memory.
- If the existing allocated memory need to extend, the pointer value will not change.
- If the existing allocated memory cannot be extended, the function allocates a new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.
- When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL

Syntax:

```
data_type *x;
x= (data_type *) realloc(p, s );
```

The size of the memory block pointed at by p changes to S. When **s > p** the additional **s-p** memory block have been extended and when **s < p**, then **p-s** bytes of the old block are freed.

Macro REALLOC

```
#define REALLOC(p,S)\
if (!((p) = realloc(p,s))) \
    { \
        fprintf(stderr, "Insufficient memory");\
        exit(EXIT_FAILURE);\
    }\
```

4. free( )

Dynamically allocated memory with either malloc( ) or calloc ( ) does not return on its own. The programmer must use free( ) explicitly to release space.

Syntax:

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated