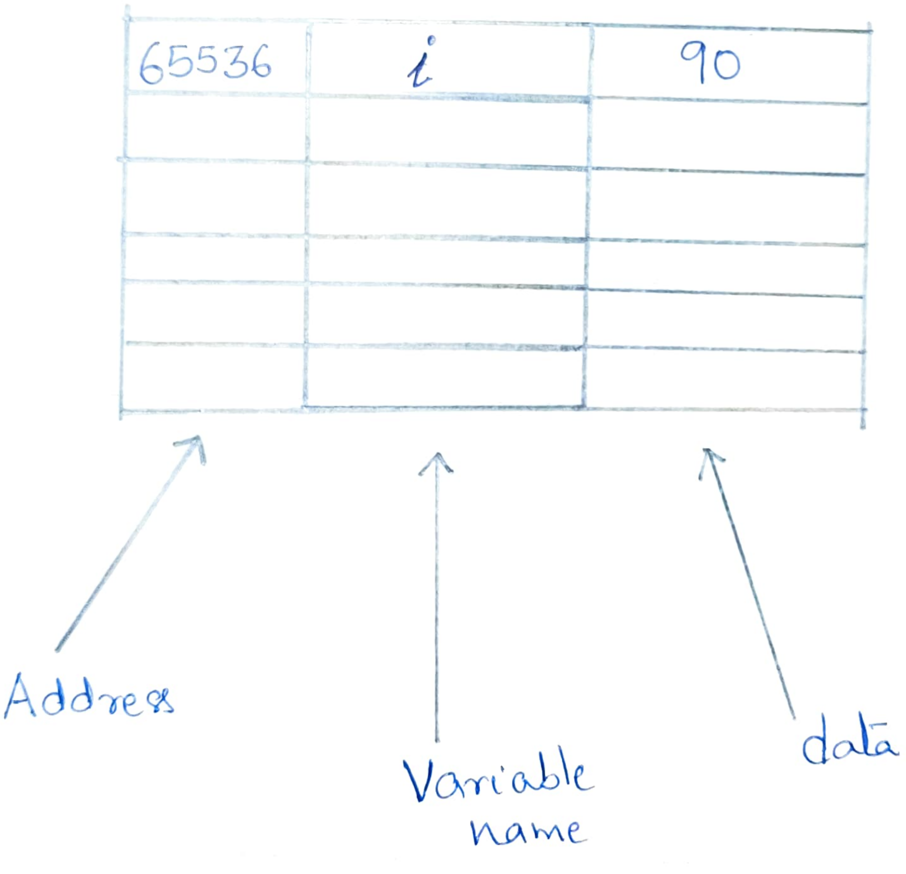
3. Pointers

Pointers are fundamental feature in C language. The computer memory is divided into number of cells. called memory location. Each memory cell is associated with fixed unique addresses. Eg: int i = 90



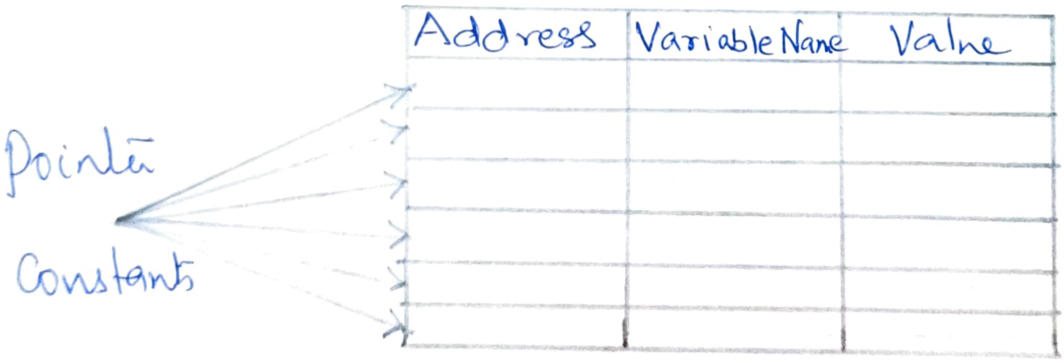
Definition: The fixed and unique addresses of memory locations are called pointers. The data stored in memory location can be of any data type such as int, float, char etc

The three important terms in pointer concept are

1. pointer constants
2. pointer values
3. pointer Variables.

1. Pointer constants:

The addresses assigned to various memory locations by operating system are unique and fixed. These addresses cannot be changed. These addresses are constants and they are called pointer constants.



2. Pointer values:

The memory is allocated to declared variable during compilation time. Each memory location allocated contains memory address. This memory address is called pointer value.

Address	Variable name	value
90000	i	10
90004	j	20
90008	k	30

For example int i= 10, j= 20, k=30;

The addresses assigned to i - 90000, j-90004, k- 90008 are called pointer values.

Pointer Variables : A variable which contains the address of the another variable is called pointer variable. For example int i=9;

Address	Variable	Value
99999	i	9

The variable which stores the address 99999 is pointer variable.

Declaration of Pointer varible in 'C'

Pointer Variable is declared in declaration Section of 'C' program. Pointer variable is variable which contain '*' prefix to it. Pointer Variable can be of any data type.

Eg.

```
int *p;
int *y;
float *q;
char *r;
double *x;
```

Two most important operators used with the pointers type are

1. & - The address operator
2. * - The dereferencing/indirection operator

If we have the declaration:

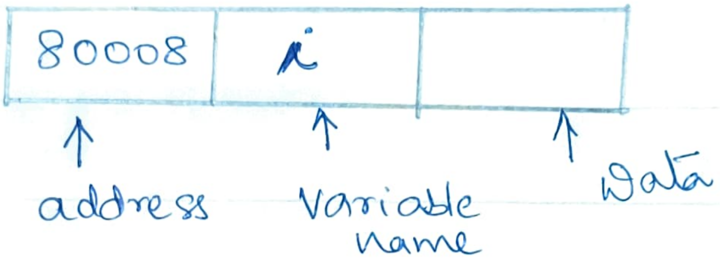
int i, *pi; then

i is an integer variable and pi is pointer to integer.

Pointer variable pi can store address of integer variable.

If we say pi = &i; // Initialization

then address of variable i is assigned to pointer variable pi



pi stores the address 80008.

To assign value to i, we can write

i = 10; or

*pi = 10;

Once address of variable is assigned to pointer variable pi. The meaning of '*' is Content of

pi contains address of i

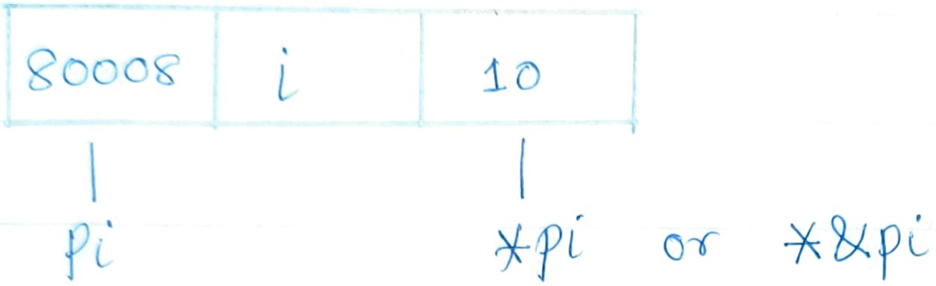
*pi means content of pi

In above example

pi contains address 80008

*pi = 10; means

Content (80008) = 10. So 10 is stored in data field.



Example:
```
Void main()
{
  int a=9; // Declaration
  int *pa;
   pa=&a; // Initialization
  Printf("%d and %d\n", *pa, a);
}
```

The output is 9 and 9

DYNMIC MEMORY ALLOCATION

The memory can be allocated to variable in two differents ways

1. static memory allocation
2. Dynamic memory allocation.

- In static memory allocation, the memory space for the variable is allocated during compilition time. Once the memory ~~space~~ space is allocated, it cannot be altered ie either expanded or reduced. during execution time.

Example : int a[100];

- During compilation, the compiler will allocate 100 locations for variable a. More than 100 elements cannot be stored because memory space cannot be altered. If the program stores only 10 elements then remaining 90 locations cannot be transferred to other program because the memory space is static cannot be altered.

To solve this problem C Programming provides mechanism called Dynamic memory allocation. In dynamic memory allocation, memory is allocated during runtime. When ever the program requires memory it can request operating system the ammount of memory required through C function malloc. If the memory is available then it is allocated to program. If requested memory is not available then pointer NULL is returned to program. At later wherever an area of memory is no longer needed by program, it can be freed by calling a function free, which returns area of memory to system.

Dynamic memory allocation functions in C

The different dynamic memory management functions that are used to allocate or deallocate memory are functions

i) malloc()

ii) Calloc()

iii) realloc()

iv) free()

malloc(): malloc() function allocates memory space during program execution time. If the memory is available by requested ammount, it returns the address of the first byte of allocated space. If requested memory space is not available then NULL is returned by malloc function.

The general form of malloc() statement is

```
ptr = (datatype *)malloc(n*sizeof(datatype));
```

For example:

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

Calloc():

Calloc() function allocate required memory size during execution time and automatically initialize memory with zeros. Calloc() returns address of the first byte of the allocated space.

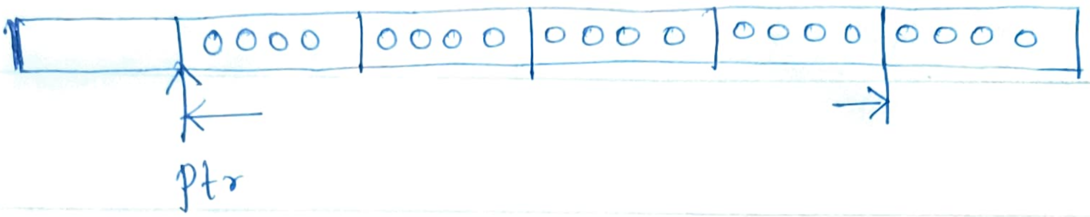
The general format of calloc() functions

```
    ptr = (datatype*) calloc(n, sizeof(datatype));
```

For example:

```
        int *ptr;
    ptr = (int *) calloc(5, sizeof(int));
```

If size of int is 4, then the memory returned is as shown below



realloc() function.

This function can be used after using malloc or calloc function. Some times allocated memory may not be sufficient or may be much larger then allocated memory can be altered using realloc().

The realloc() function.

* modifies the size of block allocated

* if existing allocated memory can be extended then ptr value will not change

* if cannot be extended then completely new block will be allocated by changing ptr address

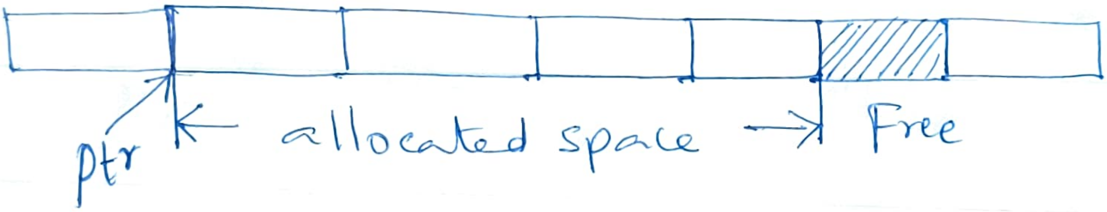
The general form is

ptr = (datatype*) realloc (ptr, n* sizeof (datatype));

The different modifications of memory space by using realloc() are

1) Reduceing the size of allocated memory

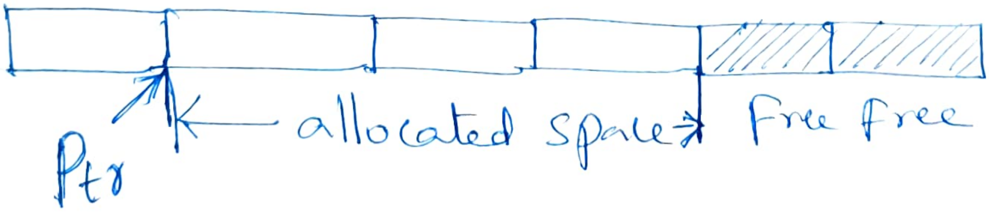
Eg: memory space already allocated is



After execution of statement,

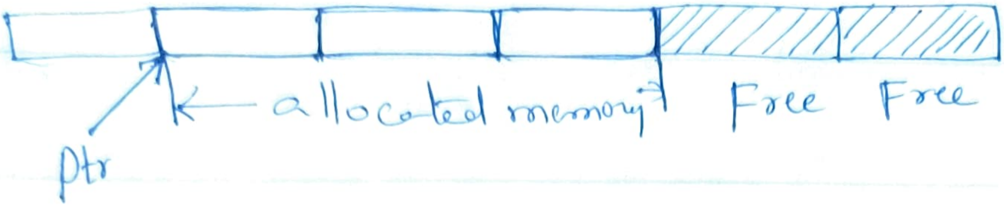
ptr = (int *) realloc (ptr, 3 * sizeof (int));

memory structure is



2) Extending the allocated memory without changing Pointer address.

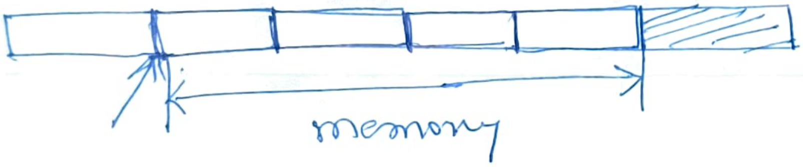
Eg: memory already allocated is



After the execution of the statement

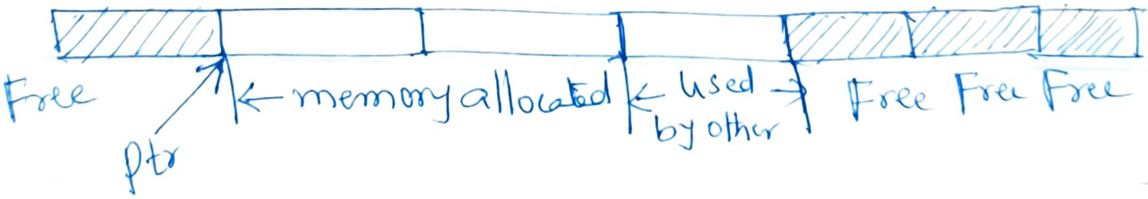
ptr= (int*) realloc(ptr, 4*Sizeof(int));

The memory structure is



3) Extending the memory by changing the address

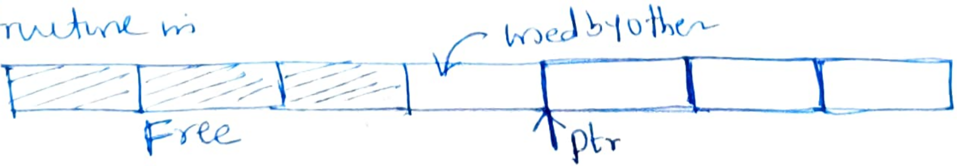
Eg: Memory already allocated is



After execution of statement

ptr= (int*)realloc(ptr, 3*Sizeof(int));

memory structure is



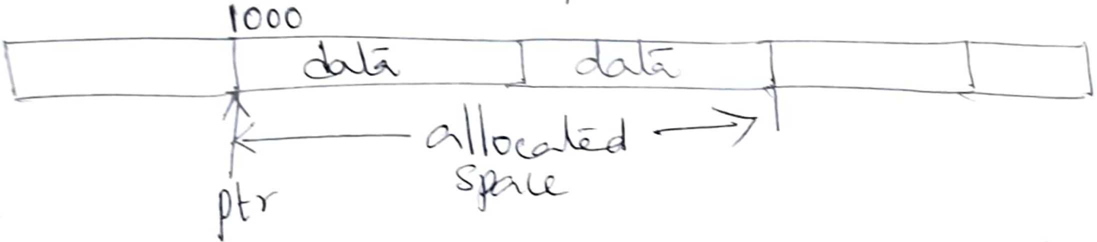
Free() function

is used to deallocate the allocated block of memory which is allocated by malloc() or calloc() function. After deallocation the pointer must be assigned with NULL.

Eg.

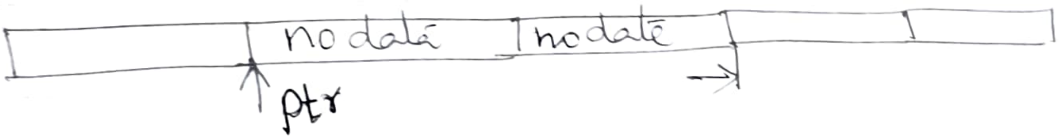
```
int *ptr;
ptr = (int*) malloc(sizeof(int));
. . . . . . . .
free(ptr);
ptr = NULL;
```

consider the following memory status



Now **ptr** is holding address 1000. After execution of statement free(ptr);

Then memory becomes



but ptr holds address 1000. So we need to assign ptr with NULL

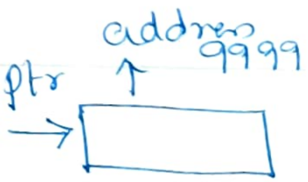
Dangling reference or Memory Leakage.

Consider the following program,

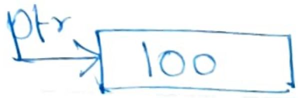
```
Void main()
  {
   int  *ptr;

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

   *ptr = 100;

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

   *ptr= 500;

  }
```

When above program is executed,

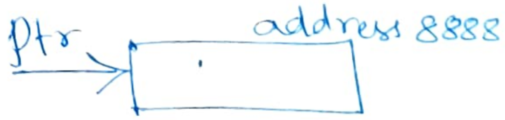
first   *ptr gets memory of size int



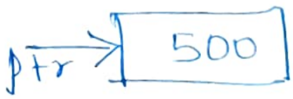
then *ptr = 100 assigns



Second   *ptr gets memory



*ptr = 500 assigns



Now ptr is pointing to memory addressed by 8888, the memory addressed by 9999 is lost and cannot be accessed by any program. This is called memory leakage or dangling pointer/reference.

Pointers can be dangerous.

The different situations, the pointer are not safe are

1) Accessing area of memory out of range of our program.

Consider the statements

```
int *p;
int a=10;

p= &a;

Printf("%d", *p);   output 10.

Printf("%d", *(p-1));   memory out of
                          range.
```

If Program tries to access the memory out of range may cause program to terminate abnormally.

2> Dereferencing the memory before allocation

Eg: Consider

```
int *p;          p [Junk address]

free(p);      Not allowed
```

Since pointer p contains junk address de-referencing p results in unpredictable output.

3> When NULL pointer is dereferenced, some computer return zero, some may return data @ in memory.

4> There is confusion since sizeof int and sizeof pointer are same, some times data may be interpreted as pointer/address.