Pointer in C

A **pointer** is a variable that stores the **memory address** of another variable. This variable can be of type int, char, array, function, or any other pointer. Instead of holding a direct value, it holds the address where the value is stored in memory. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Pointer Declaration

To declare a pointer, **dereferencing operator (*)** is used followed by the data type.

Syntax	**type *var-name;** • Where, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. • The asterisk * used to declare a pointer
Example	int *ptr; // A pointer to an integer

Operators used in Pointer

There are **2 important operators used in** pointers:

I. **Dereferencing operator / Value at address operator** (*) used to declare pointer variable and access the value stored in the address.

II. **Address operator (&)** used to returns the address of a variable or to access the address of a variable to a pointer.

Initialization

To initialize a pointer, assign it the address of a variable using the address-of operator & as:

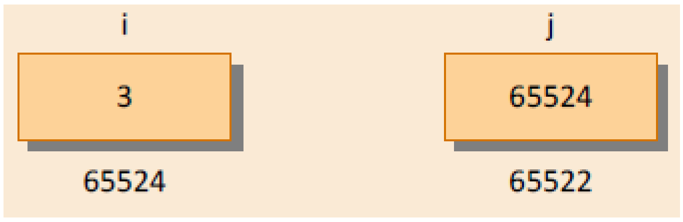
```
int var = 10;
int *ptr = &var;          // ptr now holds the address of var
```

Accessing Values

To access the value stored at the memory location pointed to by a pointer, value at address/ dereference operator * is used as

```
int value = *ptr;          // Retrieves the value of var, which is 10:
```

<u>Consider an example</u>



Example 1

```c
# include <stdio.h>
#include <conio.h>
int main( )
{
int i = 3 ;
int *j ;
j = &i ;
printf ( "Address of i = %u\n", &i ) ;
printf ( "Address of i = %u\n", j ) ;
printf ( "Address of j = %u\n", &j ) ;
printf ( "Value of j = %u\n", j ) ;
printf ( "Value of i = %d\n", i ) ;
printf ( "Value of i = %d\n", *( &i ) ) ;
printf ( "Value of i = %d\n", *j ) ;
getch() ;
}
```

Output

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

```c
# include <stdio.h>
#include <conio.h>
int main( )
{
int i = 3, *j, **k ;
j = &i ;
k = &j ;
printf ( "Address of i = %u\n", &i ) ;
printf ( "Address of i = %u\n ", j ) ;
printf ( "Address of i = %u\n ", *k ) ;
printf ( "Address of j = %u\n ", &j ) ;
printf ( "Address of j = %u\n ", k ) ;
printf ( "Address of k = %u\n ", &k ) ;
printf ( "Value of j = %u\n ", j ) ;
printf ( "Value of k = %u\n ", k ) ;
printf ( "Value of i = %d\n ", i ) ;
printf ( "Value of i = %d\n ", * ( &i ) ) ;
printf ( "Value of i = %d\n ", *j ) ;
printf ( "Value of i = %d\n ", **k ) ;
getch() ;
}
```

Output

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j = 65524
Value of k = 65522
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
```

Pointer Arithmetics in C

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers.

Operations are:

1. Increment/Decrement of a Pointer	**Increment**: When a pointer in C is incremented by1, it moves to the next memory location based on the size of the data it points to. **For example**, if you have an integer pointer and you increment it, it will jump to the next integer-sized space in memory (usually 2 or 4

	bytes). The same applies to float pointers; they move by the size of a float (typically 4 bytes) when incremented. **Decrement**: When a pointer is decremented in C, it moves backward by an amount equal to the size of the data type it is pointing to. • If an integer pointer is decremented, it moves back by 4 bytes (assuming a standard 4-byte integer on most systems). • If a float pointer is decremented, it also moves back by 4 bytes (assuming a standard 4-byte float).
2. Addition of Integer to Pointer	When an integer is added to a pointer, the integer value is first multiplied by the size of the data type to which the pointer points and then the result is added to the pointer's address. **For example:** • Consider an integer pointer storing the address 1000. • Suppose integer 5 is added to the pointer with the expression ptr = ptr + 5, the final address stored in ptr will be calculated as follows: 1000 + sizeof(int) * 5, which is equal to 1010.
3. Subtraction of Integer to Pointer	When an integer is subtracted from a pointer, the integer value is first multiplied by the size of the data type to which the pointer points and then the result is subtracted from the pointer's address. **For example:** • Consider an integer pointer storing the address 1000. • Suppose integer 5 is subtracted from it with the expression ptr = ptr - 5, the final address stored in ptr will be calculated as follows: 1000 - sizeof(int) * 5, which is equal to 990.
4. Subtraction of Two Pointers	When two pointers that point to the same data type are subtracted, the result is calculated by finding the difference between their addresses and then dividing by the size of the data type to determine how many elements (not bytes) separate the two pointers. **For example:** • Consider two integer pointers, ptr1 with an address of 1000 and ptr2 with an address of 1004. • When these two pointers are subtracted (ptr2 - ptr1), the result will be of 4 bytes, which is the difference in addresses. • Since the size of an integer is 4 bytes, you divide the address difference by the size of the data type: (4 / 4), which equals 1. This means there is an increment of 1 integer-sized element between ptr1 and ptr2.
5. Comparison of Pointers	Pointers can be compared by using operators like >, >=, <, <=, ==, and !=. These operators return true for valid conditions and false for unsatisfied conditions.

Uses of Pointers in C

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

1. Pass Arguments by Pointers
2. Accessing Array Elements
3. Dynamic Memory Allocation
4. Implementing Data Structures
5. In System-Level Programming where memory addresses are useful.

Advantages of Pointers

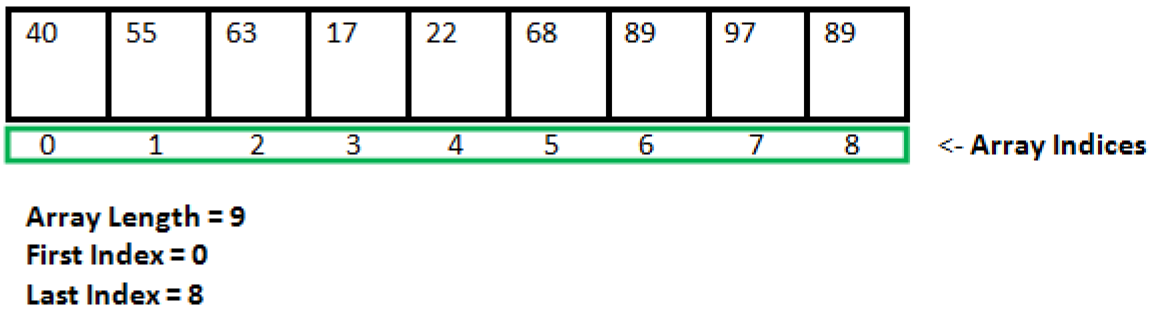
1. Pointers are used for dynamic memory allocation and deallocation.
2. An Array or a structure can be accessed efficiently with pointers
3. Pointers are useful for accessing memory locations.
4. Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
5. Pointers reduce the length of the program and its execution time as well.

Disadvantages of Pointers

1. Memory corruption can occur if an incorrect value is provided to pointers.
2. Pointers are a little bit complex to understand.
3. Pointers are majorly responsible for memory leaks in C.
4. Pointers are comparatively slower than variables in C.
5. Uninitialized pointers might cause a segmentation fault.

Dynamic Memory Allocation in C

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



As can be seen, the length (size) of the array above is 9. But what if there is a requirement to change this length (size)?

For example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

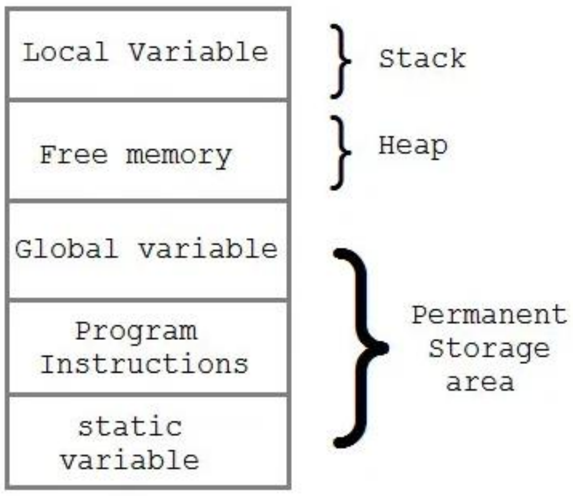
Dynamic Memory Allocation

Dynamic memory allocation in C allows to allocate memory at runtime rather than at compile time. In C language, the process of allocating memory at runtime is known as **dynamic memory allocation**. This is useful when we don't know in advance how much memory our program will need

Library functions known as **memory management functions** are used for assigning (allocating) and freeing memory, during execution of a program. These functions are defined in the **stdlib.h** header file. These functions allocate memory from a memory area known as **heap** and when the memory is not in use, it is freed to be used for some other purpose. We access the dynamically allocated memory using pointers

Memory Allocation Process in C

Global variables, static variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**. The memory space between these two regions is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keeps changing.



Dynamic memory allocation in c language is possible by 4 functions that are defined in stdlib.h header file.

1	**malloc()**	allocates single block of requested memory.
2	**calloc()**	allocates multiple blocks of requested memory.
3	**realloc()**	reallocates the memory occupied by malloc () or calloc() functions.
4	**free()**	frees the dynamically allocated memory.

1. malloc () Function

- malloc is a function in C that is used to dynamically allocate memory. It is used to request a block of memory of a specified size.
- It allows a program to allocate memory at runtime, which can be used to store data.
- This function reserves a block of memory of the given size and returns a **pointer** of type void. This means that we can assign it to any type of pointer using typecasting.
- When malloc is called, it attempts to allocate a block of memory of the specified size. If successful, it returns a void* pointer to the beginning of the allocated memory block.
- If the allocation fails (usually due to insufficient memory), it returns NULL.
- It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of malloc() in C	data_type *ptr;
	ptr = (cast-type*) malloc(byte-size);

For Example:	**ptr = (int*) malloc(100 * sizeof(int));** Since the size of int is 2 bytes, this statement will allocate 200 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

malloc() implementation in c

```
void main()
{
  int *x;
  float *y;
  x = (int*)malloc(50);     //50 contiguous bytes of memory are allocated and the address of the 
first
                                    //byte is stored in x.

  y = (float*)malloc(5 * sizeof(float);    // This allocates memory to store 5 decimal numbers.
                                                  // we should always check the value returned.
   if(x = = NULL)
   printf("Memory unavailable\n");

  if(y = = NULL)
 printf("Memory unavailable\n");

}
```

malloc() implementation in c

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
 scanf("%d",&n);

ptr=(int*)malloc(n*sizeof(int));          //memory allocated using malloc
 if(ptr= =NULL)
 {
 printf("Sorry! unable to allocate memory");
 exit(0);
}
printf("Enter elements of array: ");
 for(i=0;i<n;++i)
 {
 scanf("%d",ptr+i);
sum+=*(ptr+i);
}
  printf("Sum=%d",sum);
 free(ptr);
return 0;
}
```

2. calloc() Function

1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.
2. calloc returns a void* pointer, which needs to be cast to the correct type before use. calloc initializes all the bytes to zero, so you don't need to initialize the array elements separately.
3. The operating system checks if there is enough free memory to fulfill the request. If there is enough free memory, calloc allocates the requested amount of memory and initializes all the bytes to zero. If there is not enough free memory, calloc returns NULL to indicate that the memory allocation failed.

calloc() implementation in c

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   // Allocate memory for an array of 10 integers
   int* arr = (int*) calloc(10, sizeof(int));

   // Check if memory allocation was successful
   if (arr == NULL)
{
      printf("Memory allocation failed\n");
      return -1;
   }
   // Print the values of the array
   for (int i = 0; i < 10; i++)
 {
      printf("Value at index %d: %d\n", i, arr[i]);
   }
   // Deallocate the memory
   free(arr);

   return 0;
}
```

Difference Between Malloc and Calloc
1. malloc allocates memory but does not initialize it, while calloc allocates memory and initializes it with zeros. 2. malloc is faster than calloc because it does not need to initialize the memory.

3. realloc Function

The realloc function in C is used to change the size of a block of memory that was previously allocated using malloc, calloc or realloc. It changes the memory size.

Syntax:

```
ptr=realloc(ptr, new-size)
```

realloc implementation in C

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   // Allocate memory for an array of 5 integers
   int* arr = (int*) malloc(5 * sizeof(int));

   // Initialize the array
   for (int i = 0; i < 5; i++)
{
      arr[i] = i;
   }
   // Print the values of the array
   printf("Before reallocation:\n");
   for (int i = 0; i < 5; i++)
{
      printf("%d ", arr[i]);
   }
   printf("\n");
   // Reallocate the memory for an array of 10 integers
   arr = (int*) realloc(arr, 10 * sizeof(int));
   // Check if memory reallocation was successful
   if (arr == NULL)
{
      printf("Memory reallocation failed\n");
      return -1;
   }

   // Initialize the new elements of the array
   for (int i = 5; i < 10; i++)
{
      arr[i] = i;
   }

   // Print the values of the array
   printf("After reallocation:\n");
   for (int i = 0; i < 10; i++)
{
      printf("%d ", arr[i]);
   }
   printf("\n");

   // Deallocate the memory
   free(arr);

   return 0;
}
```

4. free() Function

The free function in C is used to dynamically de-allocate the memory that was previously allocated using malloc, calloc, or realloc.

Syntax of free() in C:

free(ptr);

free()implementation in C

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

int main() {
    // Allocate memory for an integer
    int* ptr = (int*) malloc(sizeof(int));

    // Check if memory allocation was successful
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return -1;
    }

    // Assign a value to the allocated memory
    *ptr = 10;

    // Print the value
    printf("Value: %d\n", *ptr);

    // Deallocate the memory
    free(ptr);

    // Note that ptr is no longer valid and should not be used
    return 0;
}
```

Linked List in C

A linked list is a data structure that consists of a sequence of elements where each element (or node) contains a data part and a reference (or link) to the next node in the sequence. Linked lists are dynamic in size allowing for efficient insertion and deletion of elements. **Linked List** is **chains of nodes** where each node contains information such as **data** and a **pointer to the next node** in the chain.

Linked List elements are not stored at a contiguous location. In the linked list there is a **head pointer** which points to the first element of the linked list and if the list is empty then it simply points to null or nothing.

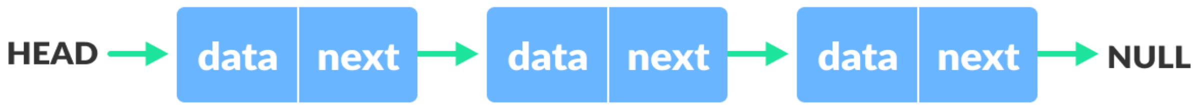
A linked list is a linear data structure where each element is known as a node that is connected to the next node using pointers. Unlike array, elements of linked list are stored in random memory locations.

A linked list is a sequence of nodes where each node contains two parts:

1. **Data**: The value stored in the node.
2. **Pointer**: A reference or link to the next node in the sequence.

A **linked list** in C is a way to store and organize data in memory using **nodes**. Each **node** contains two parts:

1. **Data** – The actual value stored.
2. **Pointer** – A link to the next node in the list.



In C, linked lists are represented as the pointer to the first node in the list. For that reason, the first node is generally called **head** of the linked list. Each node of the linked list is represented by a structure that contains a data field and a pointer of the same type as itself. Such structure is called self-referential structures.

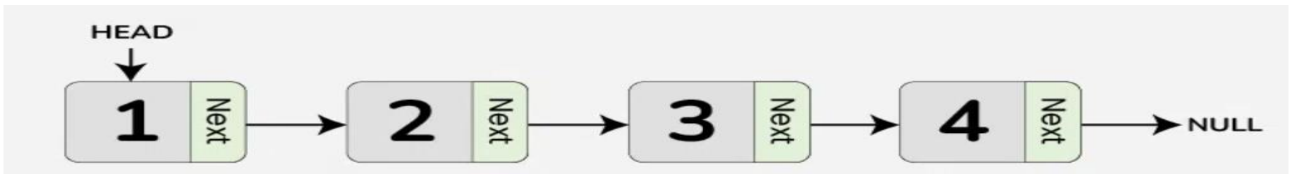
Types of Linked List in C

Linked list can be classified on the basis of the type of structure they form as a whole and the direction of access. Based on this classification, there are three types of linked lists:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

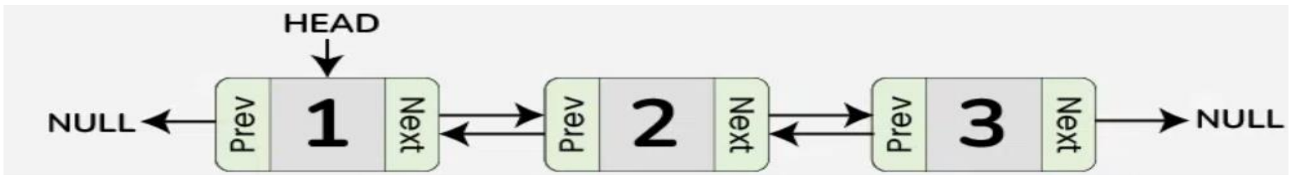
Singly Linked List in C

A linked list or singly linked list is a linear data structure that is made up of a group of nodes in which each node has two parts: the data, and the pointer to the next node. The last node's (also known as tail) pointers point to NULL to indicate the end of the linked list.



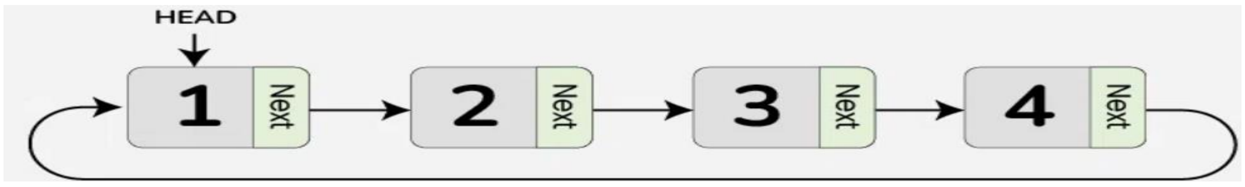
Doubly Linked List in C

A doubly linked list is a bit more complex than singly linked list. In it, each node contains three parts: the data, a pointer to the next node, and one extra pointer which points to the previous node. This allows for traversal in both directions making it more versatile than a singly linked list.



Circular Linked List in C

A circular linked list is a variation of a singly linked list where the last node points back to the first node, forming a circle. This means there is no NULL at the end and the list can be traversed in a circular manner.



Advantages of Linked Lists

- Efficient insertion and deletion of nodes
- Dynamic memory allocation
- Flexibility in data structure

Disadvantages of Linked Lists

- More memory-intensive than arrays
- Slower search times
- More complex implementation

Real-World Applications

- Google's search engine uses linked lists to store web pages in its index
- Facebook's news feed uses linked lists to store user updates
- Amazon's product catalog uses linked lists to store product information