What is Reference in c programming?

In computer science, a **reference** is a value that enables a **program** to indirectly access a particular datum, such as a variable's value or a record, in the computer's memory or in some other storage device. The **reference** is said to refer to the datum, and accessing the datum is called dereferencing the **reference**.

A reference is distinct from the datum itself. Typically, for references to data stored in memory on a given system, a reference is implemented as the physical address of where the data is stored in memory or in the storage device. For this reason, a reference is often erroneously confused with a *pointer* or *address*, and is said to "point to" the data. However, a reference may also be implemented in other ways, such as the offset (difference) between the datum's address and some fixed "base" address, as an index into an array, or more abstractly as a handle. More broadly, in networking, references may be *network* addresses, such as URLs.

Pointer

A pointer is a variable in C, and pointers value is the address of a memory location.

Pointer Definition in C

Syntax:

```
type *variable_name;
```

Example:

```
int  *width;

char  *letter;
```

Benefits of using Pointers in C

- Pointers allow passing of arrays and strings to functions more efficiently.
- Pointers make it possible to return more than one value from the function.
- Pointers reduce the length and complexity of a program.
- Pointers increase the processing speed.
- Pointers save the memory.

How to use pointers in c

Example

```
#include<stdio.h>


int main ()

{

   int  n = 20, *pntr;  /* actual and pointer variable declaration */

   pntr = &n;  /* store address of n in pointer variable*/

   printf("Address of n variable: %x\n", &n  );
```

```
   /* address stored in pointer variable */   printf("Address stored in
pntr variable: %x\n", pntr );


   /* access the value using the pointer */   printf("Value of *pntr
variable: %d\n", *pntr );


   return 0;

}
Address of n variable: 2cb60f04
```

Address stored in pntr variable: 2cb60f04

Value of *pntr variable: 20

Memory Management in C

C language provides many functions that come in header files to deal with the allocation and management of memories.

Management of Memory(static memory allocations and. dynamic memory allocations)

Management of Memory

Almost all computer languages can handle system memory. All the variables used in your program occupies a precise memory space along with the program itself, which needs some memory for storing itself (i.e., its own program). Therefore, managing memory utmost care is one of the major tasks a programmer must keep in mind while writing codes.

When a variable gets assigned in a memory in one program, that memory location cannot be used by another variable or another program. So, C language gives us a technique of allocating memory to different variables and programs.

There are two types used for allocating memory. These are:

Static memory allocations

In the static memory allocation technique, allocation of memory is done at compilation time, and it stays the same throughout the entire run of your program. Neither any changes will be there in the amount of memory nor any change in the location of memory.

Dynamic memory allocations

In dynamic memory allocation technique, allocation of memory is done at the time of running the program, and it also has the facility to increase/decrease the memory quantity allocated and can also release or free the memory as and when not required or used. Reallocation of memory can also be done when required. So, it is more advantageous, and memory can be managed efficiently.

Dynamic Memory Allocation

malloc, *calloc*, or *realloc* are the three functions used to manipulate memory. These commonly used functions are available through the *stdlib* library so you must include this library to use them.

C - Dynamic memory allocation functions

Function	Syntax
malloc()	malloc (number *sizeof(int));
calloc()	calloc (number, sizeof(int));
realloc()	realloc (pointer_name, number * sizeof(int));
free()	free (pointer_name);

malloc function

- malloc function is used to allocate space in memory during the execution of the program.
- malloc function does not initialize the memory allocated during execution. It carries garbage value.
- malloc function returns null pointer if it couldn't able to allocate requested amount of memory.

Example program for malloc() in C

Example:

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

int main()
{
char *mem_alloc;
```

```
/* memory allocated dynamically */mem_alloc = malloc( 15 * sizeof(char) );


if(mem_alloc== NULL )

{

printf("Couldn't able to allocate requested memory\n");

}

else

{

strcpy( mem_alloc,"MYPROGRAM.in");

}


printf("Dynamically allocated memory content : %s\n", mem_alloc );

free(mem_alloc);

}
```

Program Output:

```
Dynamically allocated memory content : MYPROGRAM.in
```

Calloc function

- calloc () function and malloc () function is similar. But calloc () allocates memory for zero-initializes. However, malloc () does not.

Example program for calloc() in C

Example:

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>


int main()

{

char *mem_alloc;

/* memory allocated dynamically */mem_alloc = calloc( 15, sizeof(char) );


if( mem_alloc== NULL )

{
```

```
printf("Couldn't able to allocate requested memory\n");
}
else
{
strcpy( mem_alloc,"myprogram.in");
}

printf("Dynamically allocated memory content : %s\n", mem_alloc );
free(mem_alloc);
}
```

Program Output:

```
Dynamically allocated memory content : myprogram.in
```

realloc function

- realloc function modifies the allocated memory size by malloc and calloc functions to new size.
- If enough space doesn't exist in the memory of current block to extend, a new block is allocated for the full size of reallocation, then copies the existing data to the new block and then frees the old block.

free function

- free function frees the allocated memory by malloc (), calloc (), realloc () functions.

Example program for realloc() and free()

Example:

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

int main()
{
char *mem_alloc;
/* memory allocated dynamically */mem_alloc = malloc( 20 * sizeof(char) );

if( mem_alloc == NULL )
{
printf("Couldn't able to allocate requested memory\n");
```

```
}
else
{
strcpy( mem_alloc,"myprogram.in");
}

printf("Dynamically allocated memory content  : " \ "%s\n", mem_alloc );
mem_alloc=realloc(mem_alloc,100*sizeof(char));

if( mem_alloc == NULL )
{
printf("Couldn't able to allocate requested memory\n");
}
else
{
strcpy( mem_alloc,"space is extended upto 100 characters");
}

printf("Resized memory : %s\n", mem_alloc );
free(mem_alloc);
}
```

Program Output:

```
Dynamically allocated memory content : myprogram.in
```

Resized memory: space is extended up to 100 characters