**Chapter-3**

**Pointers**

**Introduction:**

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addressed as their values. Pointers can be used to access and manipulate data stored the memory.

**Advantages of pointers:**

1. Pointers are efficient handling data associated with arrays.
2. Pointers are used to fast access the memory and for saving memory space.
3. Pointers reduce length and complexity of the program.
4. Dynamic memory allocation is possible with pointers.
5. passing arrays and structures to functions

**Understanding Pointers:** Memory Cell Address

|  |  |
| --- | --- |
|  | 0 |
|  | 1 |
|  | 2 |
|  | 3 |
|  | 4 |
|  | 5 |
|  | . |
|  | . |
|  | . |
|  | . |
|  | . |
|  | . |
|  | . |
|  | . |
|  | 65,535 |

The computer’s memory is a sequential collection

of storage cells. Each cell is commonly known as byte, has a

number called address associated with it .The addresses are

numbered consecutively, starting from zero. The last address

depends on the memory size. A computer memory having 64 K

memory will have its last address as 65,535.

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number.

Ex: int quantity = 179;

This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that

Location. Assume that system has chosen the address location **Memory Organisation**

5000 for quantity as shown below:

Quantity Variable

Value

5000 Address

179

**Representation of a Variable**

We can address to the value 179 by using either the name quantity or the address 5000. Since memory addresses are simply numbers, they can be assigned to some other variables that can be stored in memory. Such variables that hold memory addressed are called pointer variables. A **Pointer variable** is a variable which stores the address of another variable in memory.

Variable Value Address

Quantity 5000

5048

179

5000

**Pointer Variable**

**Accessing the address of the variable:**

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. The operator & immediately preceding a variable returns the address of the variable associated with it.

Ex: p = &quantiy;

Would assign the address 5000 (the location of quantity) to the variable p. The & operator can be remembered as ‘address of’.

The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. &125 (Pointing at constants).

2. int x[10];

&x (pointing at array names);

3. &(x+y) (pointing at expressions).

If x is an array, then expressions such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of x.

**Declaring Pointer variables:**

Pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

Syn: datatype \* variable\_name;

This tells the compiler three things about the variable pt\_name.

1. The asterisk (\*) tells that the variable pt\_name is a pointer variable.

2. pt\_name needs a memory location.

3. pt\_name points to a variable of type data\_type.

Ex: int \*ptr;

declares the variable ptr as a pointer variable that points to an integer data type. Here type int refers to the data type of the variable beig pointed to by p and not the type of the value of the pointer.

Similarly float \*x;

declares x as a pointer to a floating-point variable.

**Initialization of pointer variables:**

The process of assigning the address of a variable to pointer variable is known as initialization.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable.

Syn: datatype \* variable\_name= address of the variable;

Ex: int quantity;

int \*p;

p = &quantity;

We can also combine the initialization with the declaration.

Ex: int x=5;

int \* ptr=&x;

printf(“\n value of the x is : %d”,x);

printf(“\n value of the \*ptr is : %d”,\*ptr);

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

Ex: int x,\*p = &x;

Ex: int \*p = &x, x; is not valid.

We could also define a pointer variable with an initial value of NULL or 0 (zero).

Ex: int \*p = NULL;

int \*p = 0;

**Accessing a variable through its pointer:**

To access the value of the variable using the pointer, we use another unary operator \* (asterisk), usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator.

Ex: int q, \*p, n;

q = 179;

p = &q;

n = \*p;

The 1st line declares q and n as integer variables and p as a pointer variable pointing to an integer. The 2nd line assigns the value 179 to q and 3rd line assigns the address of quantity to pointer variable p. The 4th line contains the indirection operator \*. When the operator \* is placed before a pointer variable in an expression, the pointer returns the value of the variable of which the pointer value is the address. In this case, \*p returns the value of the variable quantity, because p is the address of quantity. The \* can be remembered as ‘value at address’. Thus the value of n would be 179.

The two statements p = &q;

n = \*p;

are equivalent to n = \*&q;

which in turn is equivalent to n = quantity;

In C, the assignment of pointers and addressed is always done symbolically, by means of symbolic names.

**Memory Allocation Functions:**

C gives us two types of memory allocations. They are Static Memory Allocation & Dynamic Memory Allocation.

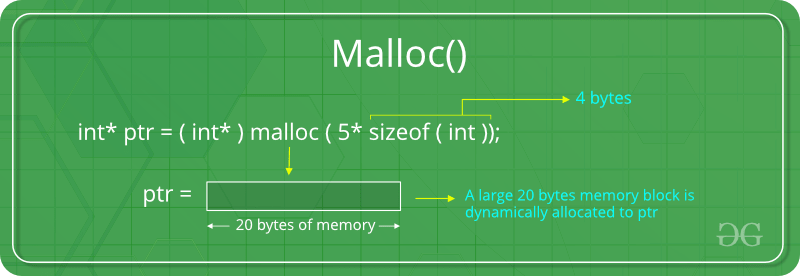
Static Memory Allocation: In static memory allocation, memory is allocated while writing the C program. This memory allocation requires that the declaration and definition of memory be fully specified in the source program. The no of bytes reserved cannot be changed during the run time. This technique can be used to define variables, arrays, pointers and streams.

Dynamic Memory Allocation: In dynamic memory allocation, memory is allocated while executing the program. That means at run time. This uses predefined functions to allocate and release memory for the data while the program is running.

Four memory management functions are used with dynamic memory. Three of them are malloc, calloc and realloc are used for memory allocation. The fourth is free, is used to return memory when it is no longer needed. All the memory management functions are found in the standard library file (stdlib.h).

Block Memory Allocation (malloc): The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of type void which can be casted into pointer of any form.

Syn: ptr = (cast-type\*) malloc(byte-size)

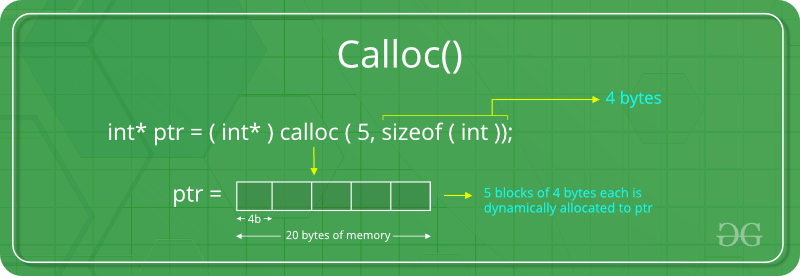


Ex: ptr = (int\*) malloc(100 \* sizeof(int));

Considering the size of int is 4 bytes, this statement allocates 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

Contiguous Memory Allocation (calloc): The malloc() function allocates a single block of memory. Whereas, calloc() allocates multiple blocks of memory and initializes them to zero.

Syn: ptr = (cast-type\*)calloc(n, element-size);

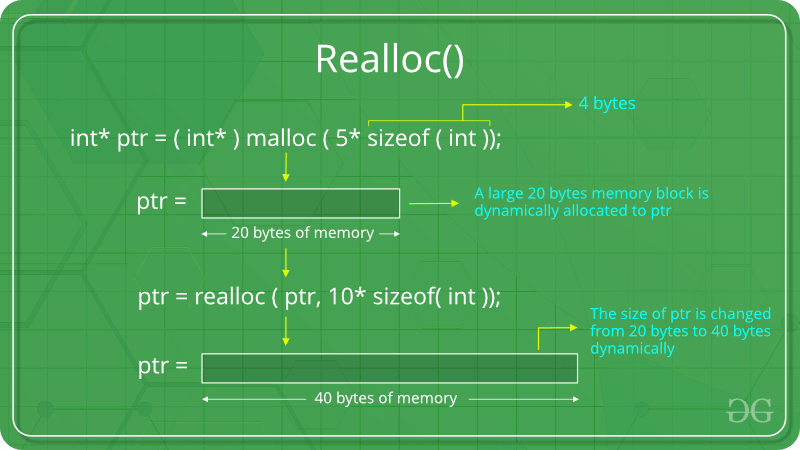


Ex: ptr = (float\*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of float.

Reallocation of Memory (realloc): If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using realloc() function.

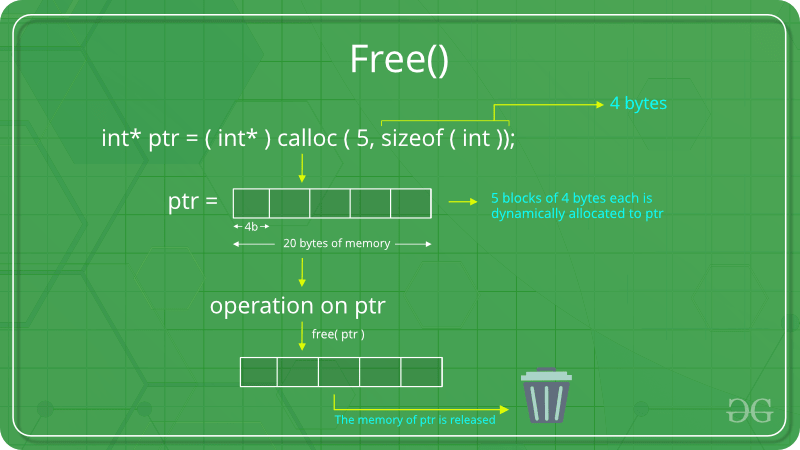
Syn: ptr = realloc(ptr, x);



Ex: realloc(ptr, 4\* sizeof(int));

Releasing Memory (free): Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syn: free(ptr);



This statement frees the space allocated in the memory pointed by ptr.