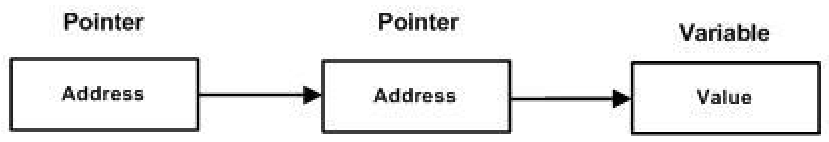
CHAIN OF POINTER:

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int −

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

Live Demo

```
#include <stdio.h>

int main () {

   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   /* take the address of var */
   ptr = &var;

   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

```c
#include <stdio.h>
int main()
{
        float var = 23.564327;

        // Declaring pointer variables upto level_4
        float *ptr1, **ptr2, ***ptr3, ****ptr4;

        // Initializing pointer variables
        ptr1 = &var;
        ptr2 = &ptr1;
        ptr3 = &ptr2;
        ptr4 = &ptr3;

        // Printing values
        printf("Value of var = %f\n", var);
        printf("Value of var using level-1"
        " pointer = %f\n",
        *ptr1);
        printf("Value of var using level-2"
        " pointer = %f\n",
        **ptr2);
        printf("Value of var using level-3"
        " pointer = %f\n",
        ***ptr3);
        printf("Value of var using level-4"
        " pointer = %f\n",
        ****ptr4);
        return 0;
}
```

Output:

```
Value of var = 23.564327
Value of var using level-1 pointer = 23.564327
Value of var using level-2 pointer = 23.564327
Value of var using level-3 pointer = 23.564327
Value of var using level-4 pointer = 23.564327
```

POINTER ARITHMETIC:

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

new_address= current_address + i * size_of(data type)

```
#include<stdio.h>

int main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p+1;

printf("After increment: Address of p variable is %u \n",p); // in our case, p will get
incremented by 4 bytes.

return 0;

}
```

Output

```
Address of p variable is 3214864300

After increment: Address of p variable is 3214864304
```

Traversing an array by using pointer

```
#include<stdio.h>

void main ()

{

        int arr[5] = {1, 2, 3, 4, 5};

        int *p = arr;

        int i;

        printf("printing array elements...\n");

        for(i = 0; i< 5; i++)

        {

        printf("%d  ",*(p+i));

        }

}
```

OUTPUT

```
printing array elements...

1  2  3  4  5
```

Decrementing Pointer in C

syntax:
new_address= current_address - i * size_of(data type)

```
#include <stdio.h>

void main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);

p=p-1;

printf("After decrement: Address of p variable is %u \n",p);

// P will now point to the immidiate previous location.

}
```

OUTPUT:

```
Address of p variable is 94343212

After decrement: Address of p variable is 94343208
```

C Pointer Subtraction

Syntax:

```
new_address= current_address - (number * size_of(data type))
```

```
#include<stdio.h>

int main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p-3; //subtracting 3 from pointer variable

printf("After subtracting 3: Address of p variable is %u \n",p);

return 0;

}
```

OUTPUT

```
Address of p variable is 1932409804
```

```
After subtracting 3: Address of p variable is 1932409792
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

new_address= current_address + (number * size_of(data type))

```
#include<stdio.h>

int main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p+3;   //adding 3 to pointer variable

printf("After adding 3: Address of p variable is %u \n",p);

return 0;

}
```

OUTPUT:

```
Address of p variable is 3911552492

After adding 3: Address of p variable is 3911552504
```

Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

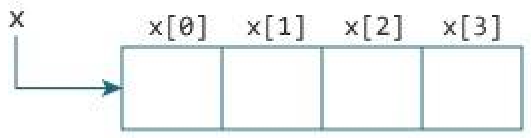
```
#include <stdio.h>

int main() {

  int x[4];

  int i;

  for(i = 0; i < 4; ++i) {

    printf("&x[%d] = %p\n", i, &x[i]);

  }

  printf("Address of array x: %p", x);

  return 0;

}
```

Output

```
&x[0] = 1450734448

&x[1] = 1450734452

&x[2] = 1450734456

&x[3] = 1450734460

Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array *x*. It is because the size of `int` is 4 bytes (on our compiler).

Notice that, the address of *&x[0]* and *x* is the same. It's because the variable name *x* points to the first element of the array.



Relation between Arrays and Pointers

From the above example, it is clear that `&x[0]` is equivalent to *x*. And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example 1: Pointers and Arrays

```
#include <stdio.h>

int main() {

  int i, x[6], sum = 0;

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {

  // Equivalent to scanf("%d", &x[i]);

      scanf("%d", x+i);

  // Equivalent to sum += x[i]

      sum += *(x+i);

  }

  printf("Sum = %d", sum);

  return 0;

}
```

OUTPUT:

```
Enter 6 numbers: 2

3

4

4

12

4

Sum = 29
```

Here, we have declared an array *x* of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that **pointers and arrays are not the same**.

There are a few cases where array names don't decay to pointers.

Example 2: Arrays and Pointers

```
#include <stdio.h>

int main() {

  int x[5] = {1, 2, 3, 4, 5};

  int* ptr;

  // ptr is assigned the address of the third element

  ptr = &x[2];

  printf("*ptr = %d \n", *ptr);   // 3

  printf("*(ptr+1) = %d \n", *(ptr+1)); // 4

  printf("*(ptr-1) = %d", *(ptr-1));  // 2

  return 0;

}
```

OUTPUT:

```
*ptr = 3

*(ptr+1) = 4

*(ptr-1) = 2
```

In this example, `&x[2]`, the address of the third element, is assigned to the *ptr* pointer. Hence, `3` was displayed when we printed `*ptr`.

And, printing `*(ptr+1)` gives us the fourth element. Similarly, printing `*(ptr-1)` gives us the second element.

Then, the elements of the array are accessed using the pointer notation. By the way,

- `data[0]` is equivalent to `*data` and `&data[0]` is equivalent to `data`
- `data[1]` is equivalent to `*(data + 1)` and `&data[1]` is equivalent to `data + 1`
- `data[2]` is equivalent to `*(data + 2)` and `&data[2]` is equivalent to `data + 2`
- `...`
- `data[i]` is equivalent to `*(data + i)` and `&data[i]` is equivalent to `data + i`

Array of pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers −

Live Demo

```
#include<stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};

 int i;

  for (i = 0; i < MAX; i++) {

    printf("Value of var[%d] = %d\n", i, var[i] );
  }

  return 0;

}
```

OUTPUT:

```
Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200
```

Array of pointer program:

```
#include <stdio.h>

int main () {

  int  var[] = {10, 100, 200};

  int i, *ptr[3];

  for ( i = 0; i < 3; i++) {

        ptr[i] = &var[i]; /* assign the address of integer. */

  }

  for ( i = 0; i < 3; i++) {

        printf("Value of var[%d] = %d\n", i, *ptr[i] );

  }

  return 0;

}
```

OUTPUT:

```
Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200
```

```
#include <stdio.h>

int main () {

  char *names[] = {

    "Zara Ali",

    "Hina Ali",

    "Nuha Ali",

    "Sara Ali"

  };

  int i = 0;

  for ( i = 0; i < 4; i++) {

    printf("Value of names[%d] = %s\n", i, names[i] );

  }

  return 0;

}
```

OUTPUT:

```
Value of names[0] = Zara Ali

Value of names[1] = Hina Ali

Value of names[2] = Nuha Ali

Value of names[3] = Sara Ali
```

Creating a String

Individual characters in C are enclosed within a single quote for example, 'a', 'b', 'c'. As explained in the previous section, a string is a collection of characters. To store a string in C, we can create an array and store them in these arrays.

Syntax

To store a string in an array, we need to declare a one-dimensional array. The characters in the string can be set at the time of array declaration or later by accessing individual indexes, as shown below.

char array_name[array_size] = {'a', 'b', ...};

// OR

char array_name[array_size];

array_name[0] = 'a';

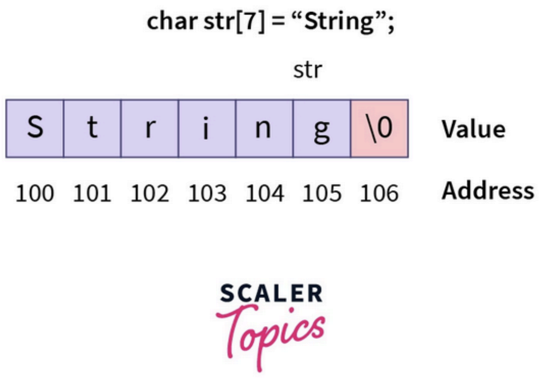
array_name[1] = 'b';

...

For example, to store a string "String" in an array str

char str[7] = {'S', 't', 'r', 'i', 'c', 'g', '\0'}; // Stricg

str[4] = 'n'; // String



Notice that although our string "String" has only six characters, our str array is of size 7 (one more than size) to store an extra null \0 character, so the value of our str array is "String\0". Also, as shown in figure **, each character takes 1-byte** of the memory space.

Note: We don't have to explicitly add a null character in the string as the compiler automatically adds it.

We can also use string literal to set the value of arrays. A **string literal** is a sequence of characters enclosed in double quotation marks (**" "**). The below-mentioned example is a simple string literal

/* string literal */

char *string_literal = "This is a string literal."

The compiler automatically adds an extra null character at the end of the string if it is not mentioned explicitly.

Instead of initializing each character individually, string literal can be used to set the value of character array like mention below

char str1[7] = "String"; /* \0 not explicitly mentioned */

// OR

char str2[7] = "String\0";

What happens if space for a null character is not allocated?

What will happen if we accidentally overwrite the null terminator in a string or try to do something like

char foo[3] = {'f', 'o', 'o'};

If a string doesn't have a null character, the compiler will still let the program pass without any error. A string is simply a collection of characters, and we need \0 only to identify the end of the string.

Having no terminator in your c-string will result in functions on the string not being able to determine the end of the string, which will cause some undefined behaviour. If we try to print the string, it might crash due to a segmentation fault, or maybe it reads random chars from memory that is located after the string until it eventually finds a null character.

Creating a Pointer for the String

When we create an array, the variable name points to the address of the first element of the array. The other way to put this is the variable name of the array points to its starting position in the memory.

We can create a character pointer to string in C that points to the starting address of the character array. This pointer will point to the starting address of the string, that is **the first character of the string**, and we can dereference the pointer to access the value of the string.

// character array storing the string 'String'

char str[7] = "String";

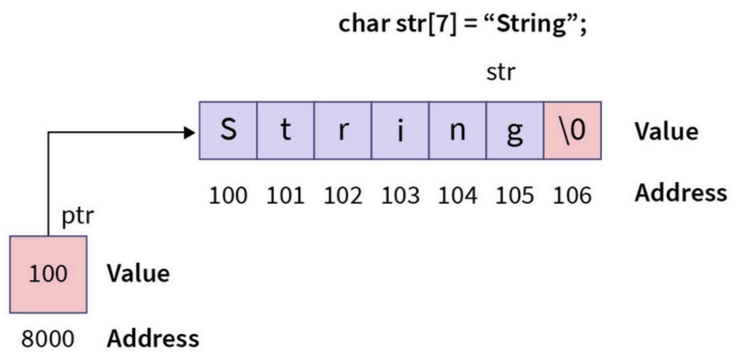
// pointer storing the starting address of the

// character array str

char *ptr = str;

In this code mentioned above, the character pointer to string in C ptr points to the starting address of the array str.

Note: Pointer and array are not the same, and here pointer stores the starting address of the array, and it can be dereferenced to access the value stored in the address.



ptr pointer points to starting address of pointer str that holds the string

We can also understand this from the figure here: pointer ptr stores the location of the first index of the array str at memory location 1000, but the pointer itself is located at memory address 8000.

Accessing String via a Pointer

An array is a contiguous block of memory, and when pointers to string in C are used to point them, the pointer stores the starting address of the array. Similarly, when we point a char array to a pointer, we pass the base address of the array to the pointer. The pointer variable can be dereferenced using the asterisk * symbol in C to get the character stored in the address. For example,

char arr[] = "Hello";

// pointing pointer ptr to starting address

// of the array arr

```c
char *ptr = arr;
```

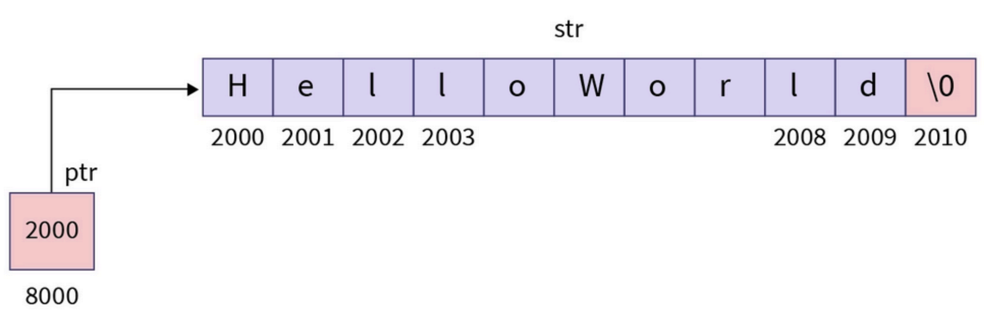
In this case, ptr points to the starting character in the array arr i.e. H. To get the value of the first character, we can use the * symbol, so the value of *ptr will be H. Similarly, to get the value of ith character, we can add i to the pointer ptr and dereference its value to get ith character, as shown below.

```c
printf("%c ", *ptr);        // H

printf("%c ", *(ptr + 1)); // e

printf("%c ", *(ptr + 2)); // l

printf("%c ", *(ptr + 3)); // l

printf("%c ", *(ptr + 4)); // o
```

Instead of incrementing the pointer manually to get the value of the string, we can use a simple fact that our string terminates with a null \0 character and use a while loop to increment the pointer value and print each character till our pointer points to a null character.

Let us understand this with an example.

```c
#include<stdio.h>

int main() {

        // creating a character array to store the value of

        // our string, notice the size of array is

        // 11 = length("HelloWorld") + 1

        char str[11] = "HelloWorld";

        // pointer variable

        char *ptr = str;

        // creating a while loop till we don't find

        // a null character in the string

        while (*ptr != '\0') {

        // the current character is not \0

        // so we will print the character
        printf("%c", *ptr);

        // move to the next character.

        ptr++;

        }

        return 0;

}
```



ptr pointer points to starting address of pointer str that holds the string

OUTPUT

```
HelloWorld
```

In the above example, we have created a character pointer to a string in C that points to the first address of the array str. To print the value stored in the array, we create a while loop until the value at the location pointed by ptr is not null, which indicates that we have not reached the end of the string. After printing the current character, we increment the ptr pointer to move to the following location. The loop terminates when we reach the null character indicating the end of the string.

Using a Pointer to Store String

Arrays are essentially continuous blocks in the memory; we can also store our strings using pointers and can dereference the pointer variables to access the value of the string. To store the string in a pointer variable, we need to create a char type variable and use the asterisk * operator to tell the compiler the variable is a pointer. This can be understood from the example,

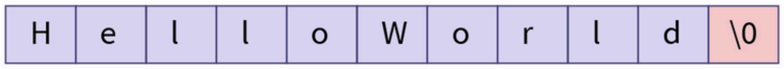
```
// storing string using an array
```

```
char arr[] = "ThisIsString\0";



// storing string using a pointer

char *str  = "ThisIsString\0";
```

Asterisk operator * can be used to access the ith character of the string that is the value of ith string character will be *(str + i).

Let us understand this with the example where we are using the pointer variable strPtr to store the string value.

```
#include<stdio.h>

int main() {

    // creating a pointer variable to store the value of

    // our string

    char *strPtr = "HelloWorld";

    // temporary pointer to iterate over the string

    char *temp = strPtr;

    // creating a while loop till we don't find

    // a null character in the string

    while (*temp != '\0') {

    // the current character is not \0

    // so we will print the character

    printf("%c", *temp);

    // move the temp pointer to the next memory location

    temp++;

    }

    return 0;
}
```



OUTPUT

```
HelloWorld
```

Here, we are using a temporary variable temp, to print the characters of the string because we don't want to lose the starting position of our string by incrementing the pointer strPtr inside the loop.

So at the end of the code, the pointer temp will point to the last character in the string *″ HelloWorld\0″* that is null (\0) but our main pointer strPtr still points to the location of the first character in the string.

Passing Pointers to Functions in C

Passing the pointers to the function means the memory location of the variables is passed to the parameters in the function, and then the operations are performed. The function definition accepts these addresses using pointers, addresses are stored using pointers.

Arguments Passing without pointer

When we pass arguments without pointers the changes made by the function would be done to the local variables of the function.

Below is the C program to pass arguments to function without a pointer:

```c
// C program to swap two values

//  without passing pointer to
// swap function.
#include <stdio.h>

void swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}

// Driver code
int main()
{
  int a = 10, b = 20;
  swap(a, b);
  printf("Values after swap function are: %d, %d", a, b);
  return 0;
}
```

Output

```
Values after swap function are: 10, 20
```

Arguments Passing with pointers

A pointer to a function is passed in this example. As an argument, a pointer is passed instead of a variable and its address is passed instead of its value. As a result, any change made by the function using the pointer is permanently stored at the address of the passed variable. In C, this is referred to as call by reference.

Below is the C program to pass arguments to function with pointers:

```
// C program to swap two values

// without passing pointer to
// swap function.
#include <stdio.h>

void swap(int* a, int* b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
int main()
{
  int a = 10, b = 20;
  printf("Values before swap function are: %d, %d\n", a, b);
  swap(&a, &b);
  printf("Values after swap function are: %d, %d", a, b);
  return 0;
}
```

Output

```
Values before swap function are: 10, 20

Values after swap function are: 20, 10
```

How to return a Pointer from a Function in C

Pointers in C programming language is a variable which is used to store the memory address of another variable. We can pass pointers to the function as well as return pointer from a function. But it is not recommended to return the address of a local variable outside the function as it goes out of scope after function returns.

C also allows to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example −

int * myFunction() {

.

.

.

}

Second point to remember is that, it is not a good idea to return the address of a local variable outside the function, so you would have to define the local variable as **static** variable.

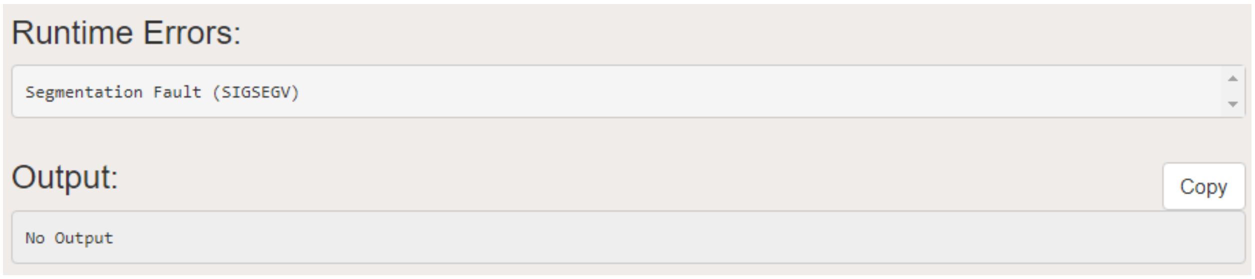
Program 1:

The below program will give segmentation fault since **'A'** was local to the function:

```
// C program to illustrate the concept of

// returning pointer from a function
#include <stdio.h>

// Function returning pointer
int* fun()
{
    int A = 10;
    return (&A);
}

// Driver Code
int main()
{
    // Declare a pointer
    int* p;

    // Function call
    p = fun();

    printf("%p\n", p);
    printf("%d\n", *p);
    return 0;
}
```

Output:

Below is the output of the above program:



```
Segmentation Fault (SIGSEGV)
```

```
No Output
```

Explanation:

The main reason behind this scenario is that compiler always make a stack for a function call. As soon as the function exits the function stack also gets removed which causes the local variables of functions goes out of scope.

Static Variables have a property of preserving their value even after they are out of their scope. So to execute the concept of returning a pointer from function in C you must define the local variable as a static variable.

Program 2:

```
// C program to illustrate the concept of

// returning pointer from a function
#include <stdio.h>

// Function that returns pointer
int* fun()
{
    // Declare a static integer
    static int A = 10;
    return (&A);
}

// Driver Code
int main()
{
    // Declare a pointer
    int* p;

    // Function call
    p = fun();

    // Print Address
    printf("%p\n", p);

    // Print value at the above address
    printf("%d\n", *p);
    return 0;
}
```

Output:

```
0x601038

10
```

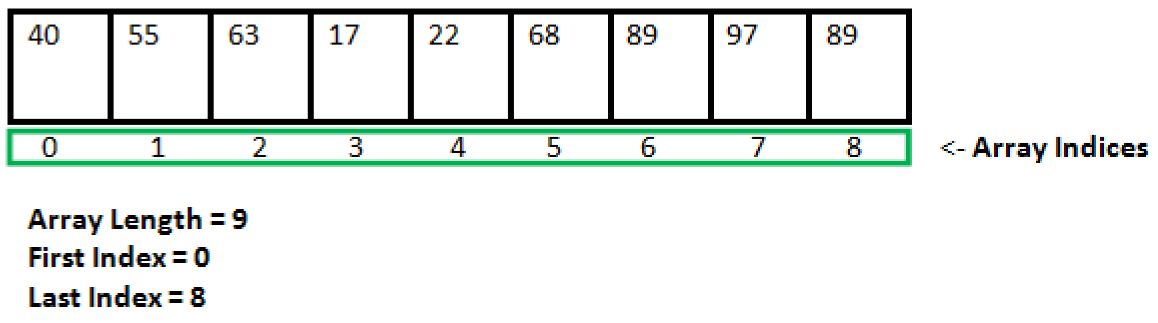
C Dynamic Memory Allocation

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

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 it can be seen that the length (size) of the array above made 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.

This procedure is referred to as **Dynamic Memory Allocation in C**.
Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

C malloc() method

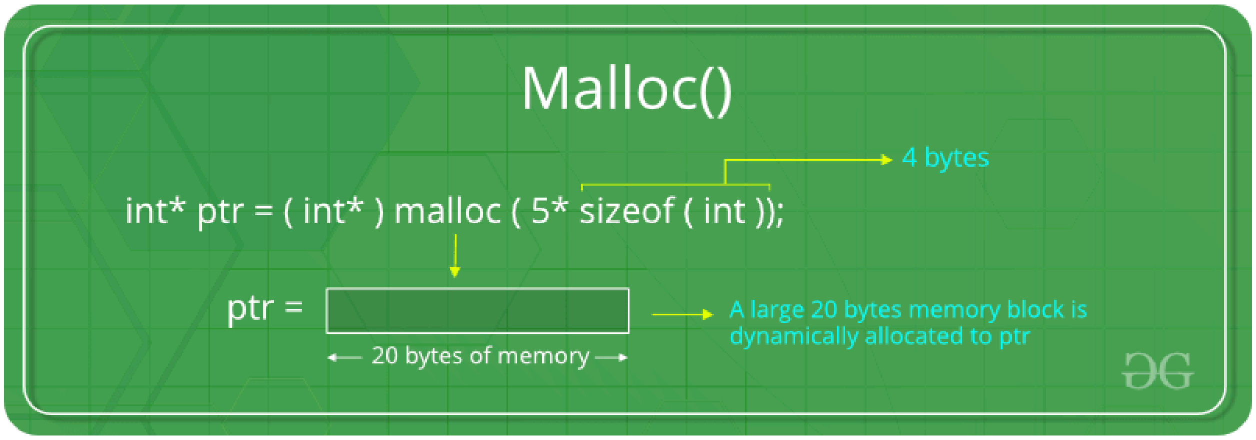
The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of malloc() in C

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));
Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

If space is insufficient, allocation fails and returns a NULL pointer.

C calloc() method

1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

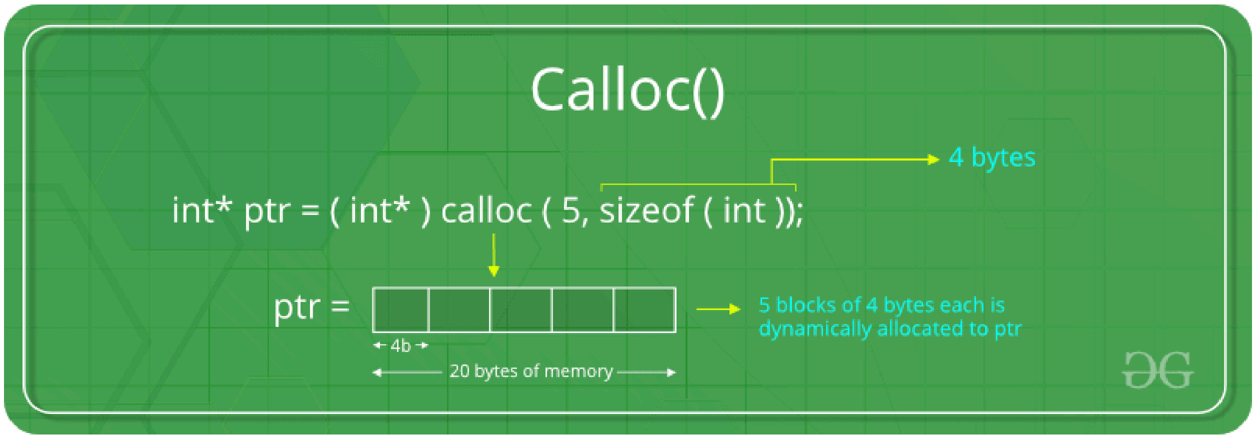
Syntax of calloc() in C

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

here, n is the no. of elements and element-size is the size of each element.

For Example:

ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for 25 elements each with the size of the float.


Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory

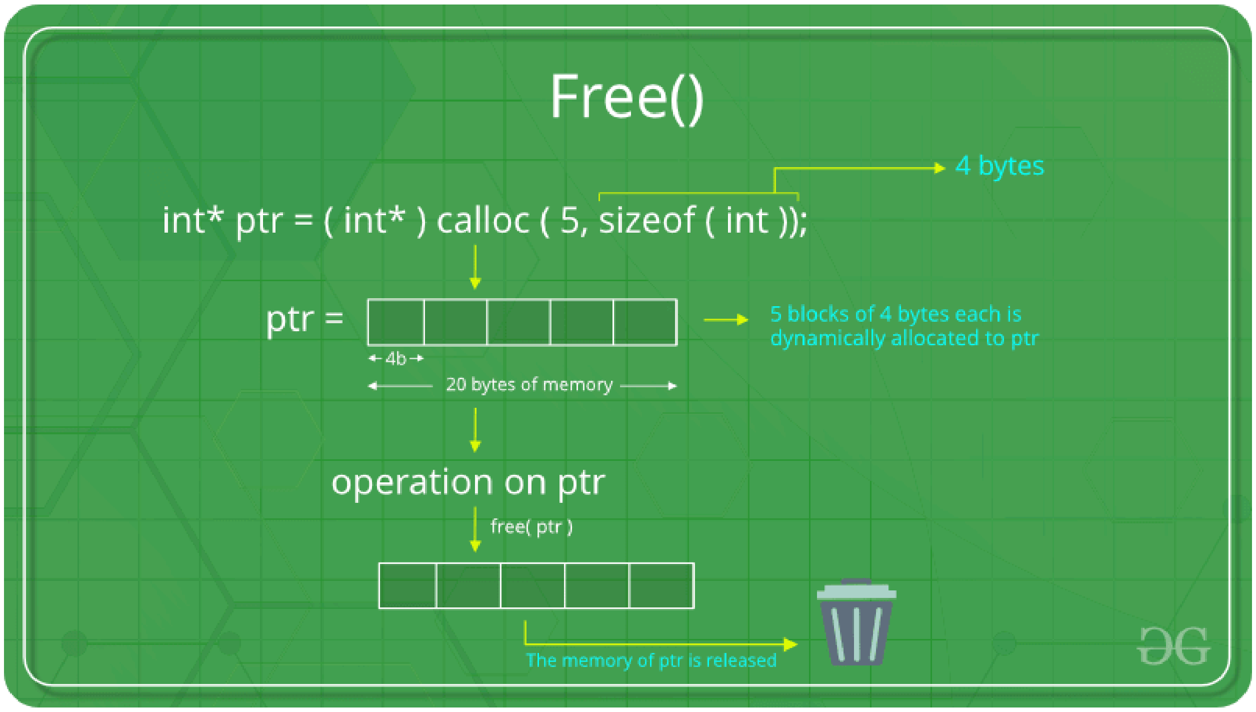
If space is insufficient, allocation fails and returns a NULL pointer.

C free() method

“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

free(ptr);


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released

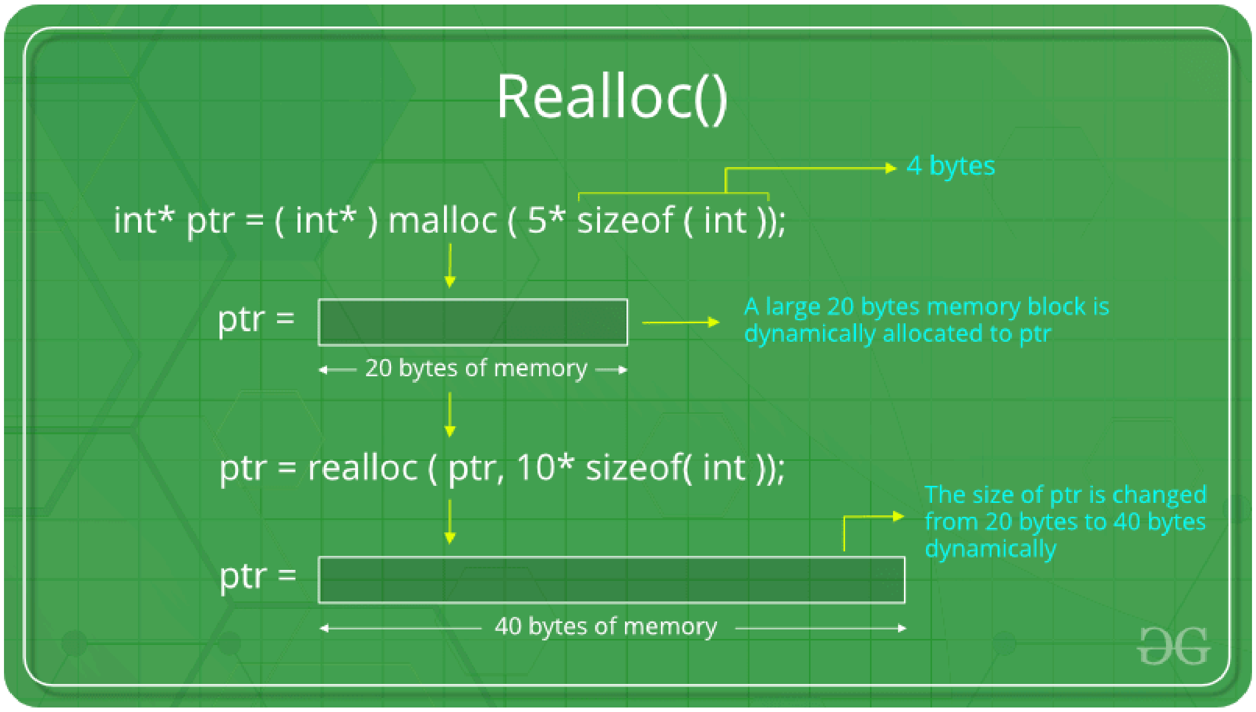
C realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

If space is insufficient, allocation fails and returns a NULL pointer.