C OPERATORS

Operators are used to perform operations on variables and values.

In the example below, we use the **+** operator to add together two values:

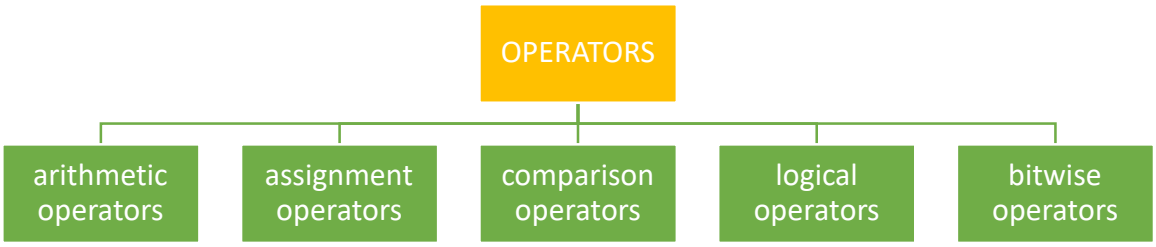
Example

```
int myNum = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50;       // 150 (100 + 50)
int sum2 = sum1 + 250;     // 400 (150 + 250)
int sum3 = sum2 + sum2;    // 800 (400 + 400)
```



Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example	Try it
+	Addition	Adds together two values	x + y	Try it »

-	Subtraction	Subtracts one value from another	x - y	Try it »
*	Multiplication	Multiplies two values	x * y	Try it »
/	Division	Divides one value by another	x / y	Try it »
%	Modulus	Returns the division remainder	x % y	Try it »
++	Increment	Increases the value of a variable by 1	++x	Try it »
--	Decrement	Decreases the value of a variable by 1	--x	Try it »

Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:

Operator	Example	Same As	Try it
=	x = 5	x = 5	Try it »
+=	x += 3	x = x + 3	Try it »
-=	x -= 3	x = x - 3	Try it »
*=	x *= 3	x = x * 3	Try it »

/=	x /= 3	x = x / 3	Try it »
%=	x %= 3	x = x % 3	Try it »
&=	x &= 3	x = x & 3	Try it »
|=	x |= 3	x = x | 3	Try it »
^=	x ^= 3	x = x ^ 3	Try it »
>>=	x >>= 3	x = x >> 3	Try it »
<<=	x <<= 3	x = x << 3	Try it »

Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either 1 or 0, which means **true** (1) or **false** (0). These values are known as **Boolean values**.

In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:

Example

```
int x = 5;
int y = 3;
printf("%d", x > y); // returns 1 (true) because 5 is greater than 3
```

A list of all comparison operators:

Operator	Name	Example	Description	Try it
==	Equal to	x == y	Returns 1 if the values are equal	Try it »
!=	Not equal	x != y	Returns 1 if the values are not equal	Try it »

>	Greater than	x > y	Returns 1 if the first value is greater than the second value	Try it »
<	Less than	x < y	Returns 1 if the first value is less than the second value	Try it »
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value	Try it »
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value	Try it »

Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values, by combining multiple conditions:

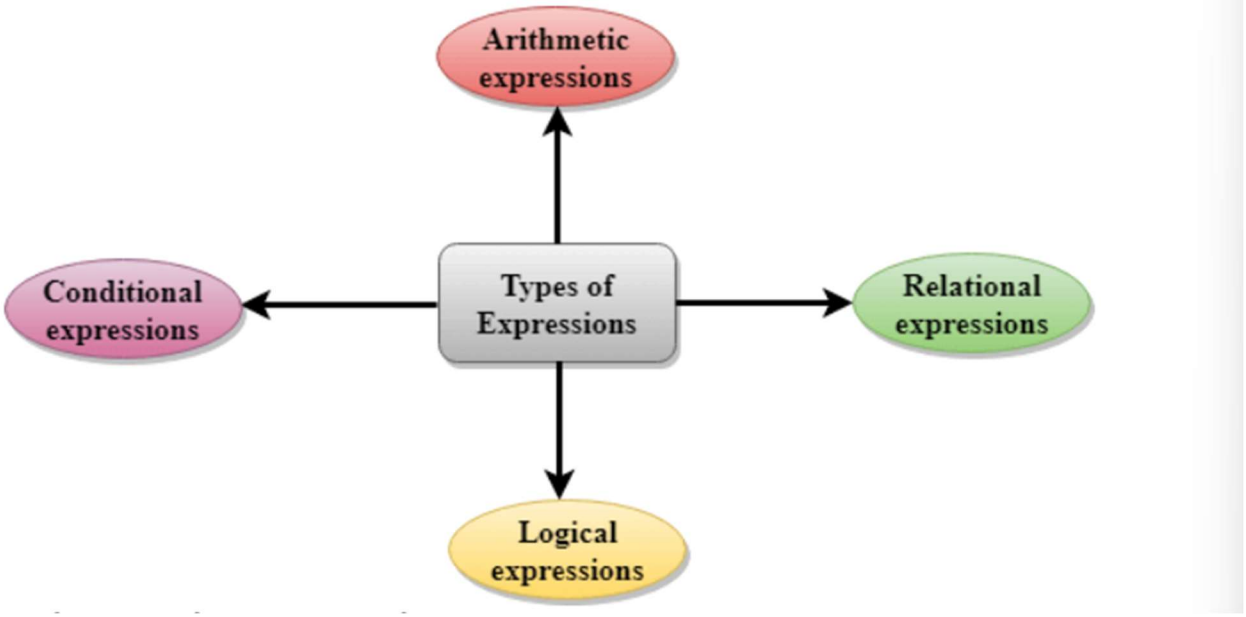
Operator	Name	Example	Description	Try it
&&	AND	x < 5 && x < 10	Returns 1 if both statements are true	Try it »
||	OR	x < 5 || x < 4	Returns 1 if one of the statements is true	Try it »
!	NOT	!(x < 5 && x < 10)	Reverse the result, returns 0 if the result is 1	Try it »

C Expressions

An expression is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

Example:

```
a-b;

x = 9/2 + a-b;
```



C If ... Else

Conditions and If Statements

You have already learned that C supports the usual logical **conditions** from mathematics

we can use these conditions to perform different actions for different decisions.

C has the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use **else if** to specify a new condition to test, if the first condition is false
- Use **switch** to specify many alternative blocks of code to be executed

The if Statement

Use the if statement to specify a block of code to be executed if a condition is true.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

C Else

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax

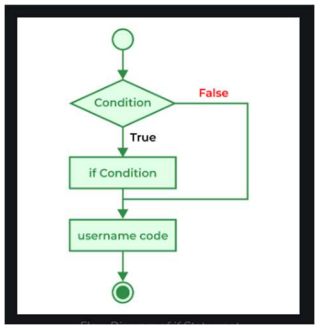
```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

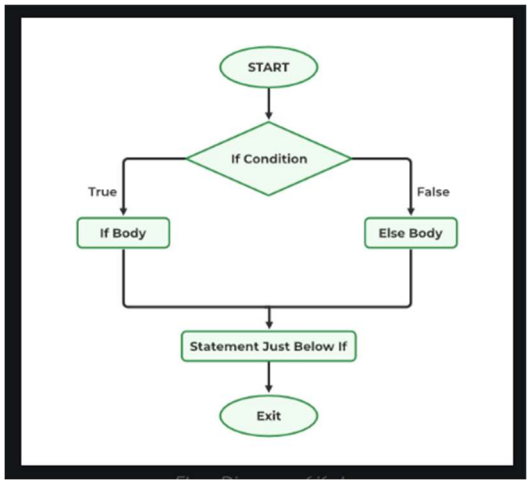
C Else If

Use the else if statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```





C Short Hand If Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Instead of writing:

Example

```
int time = 20;
if (time < 18) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
```

You can simply write:

Example

```
int time = 20;
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

C Switch

Instead of writing **many** if..else statements, you can use the switch statement.

The switch statement selects one of many code blocks to be executed:

Syntax

```
switch (expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

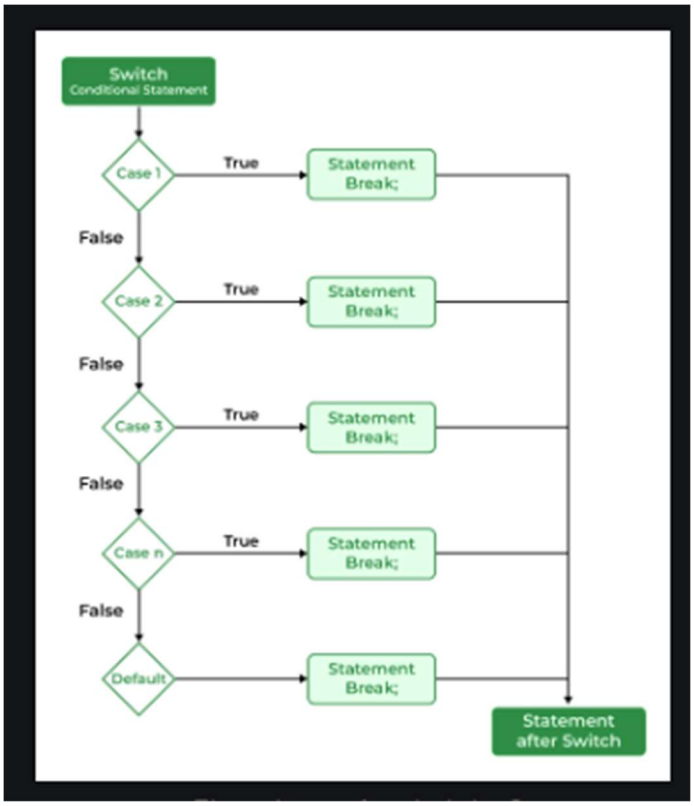
- The switch expression is evaluated once
- The value of the expression is compared with the values of each case

- If there is a match, the associated block of code is executed
- The break statement breaks out of the switch block and stops the execution
- The default statement is optional, and specifies some code to run if there is no case match

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
  case 5:
    printf("Friday");
    break;
  case 6:
    printf("Saturday");
    break;
  case 7:
    printf("Sunday");
    break;
}

// Outputs "Thursday" (day 4)
```



Using Range in switch Case in C

you can use a range of numbers instead of a single number or character in the case statement.

Range in switch case can be useful when we want to run the same set of statements for a range of numbers so that we do not have to write cases separately for each value.

Syntax

The syntax for using range case is:

```
case low ... high:
```

It can be used for a range of ASCII character codes like this:

```
case 'A' ... 'Z':
```

You need to Write spaces around the ellipses **(...).** For example, write this:

```
// Correct  -   case 1 ... 5:
// Wrong -    case 1...5:
```

```c
// C program to illustrate
// using range in switch case
#include <stdio.h>
int main()
{
    int arr[] = { 1, 5, 15, 20 };

    for (int i = 0; i < 4; i++) {
        switch (arr[i]) {
            // range 1 to 6
        case 1 ... 6:
            printf("%d in range 1 to 6\n", arr[i]);
            break;
            // range 19 to 20
        case 19 ... 20:
            printf("%d in range 19 to 20\n", arr[i]);
            break;
        default:
            printf("%d not in range\n", arr[i]);
            break;
        }
    }
    return 0;
}
```

Jump Statements in C

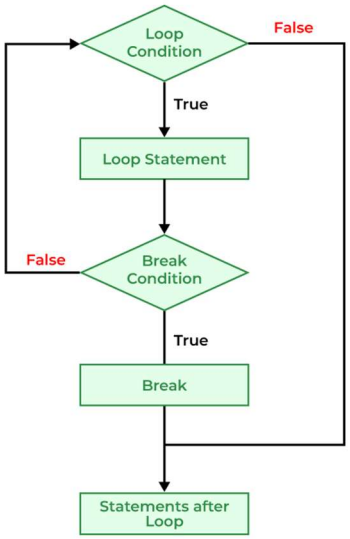
These statements are used in C for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

D) break

This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Syntax of break

```
break;
```



B) continue

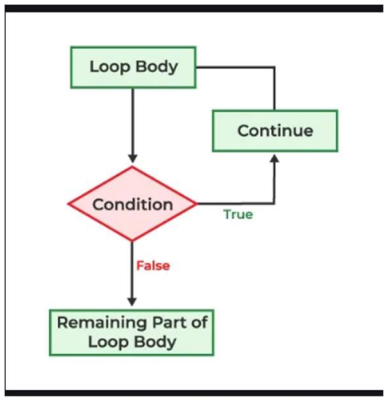
The *continue* statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.
When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

Syntax of continue

```
continue;
```

Flowchart of Continue

Flow Diagram of C continue Statement



C) goto

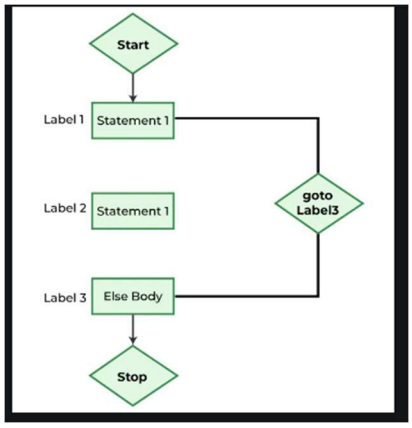
The goto statement in C also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Syntax of goto

Syntax1	**Syntax2**
goto *label;*	*label:*
.	.
.	.
.	.
label:	**goto** *label;*

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, a label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.

Flowchart of goto Statement



Examples of goto

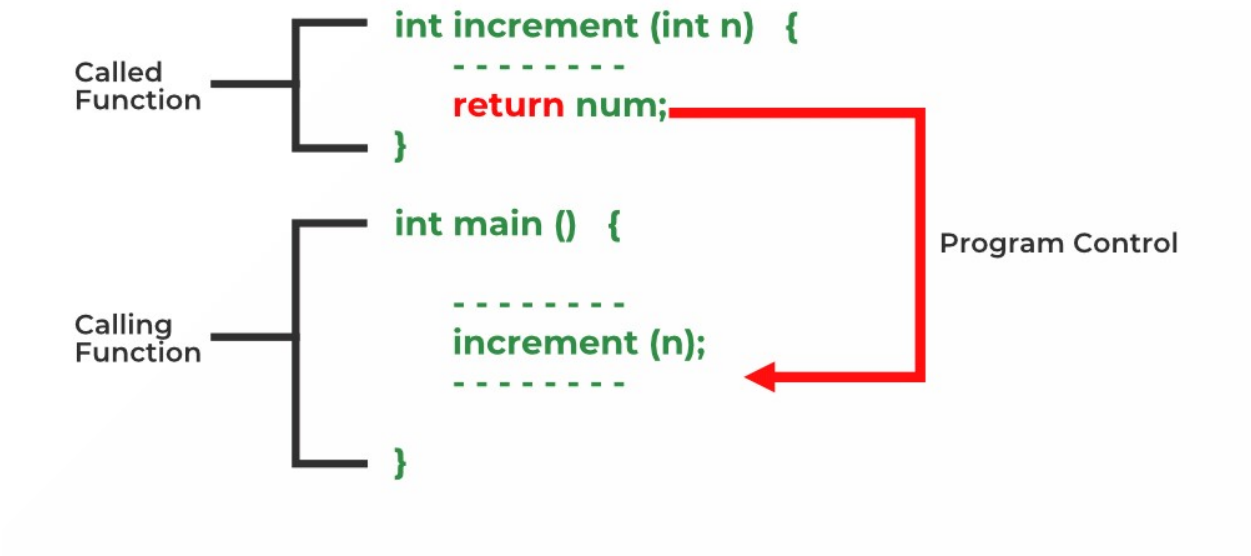
C

```c
// C program to print numbers
// from 1 to 10 using goto
// statement
#include <stdio.h>

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
int main()
{
    printNumbers();
    return 0;
}
```

D) return

The return in C returns the flow of the execution to the function from where it is called. As soon as the statement is executed, the flow of the program stops immediately. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

Flowchart of return



Flow Diagram of return

Syntax of return

return *[expression]*;

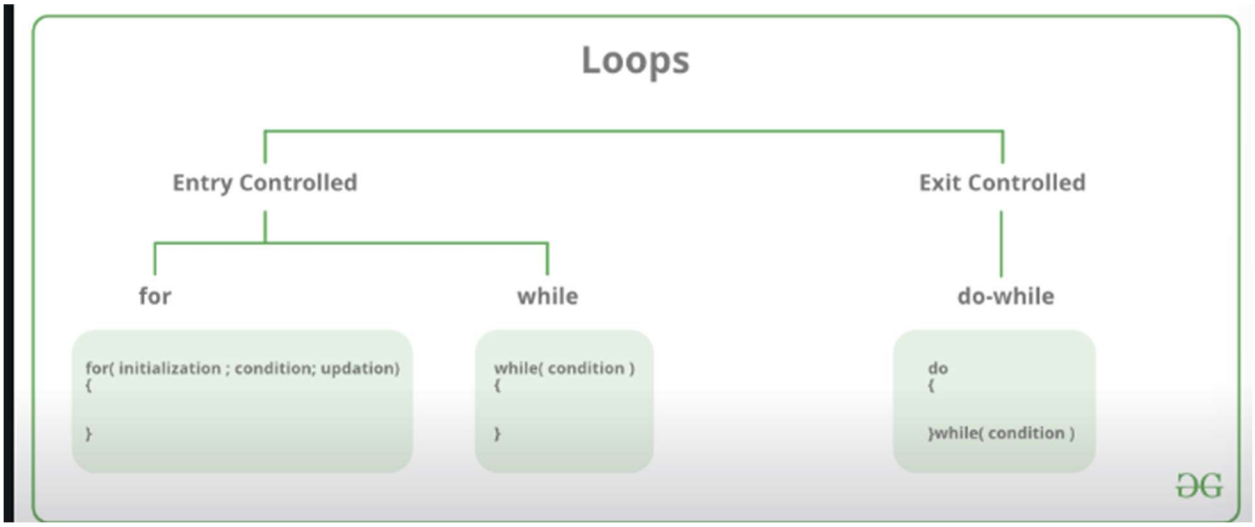
C – Loops

Loops in programming are used to repeat a block of code until the specified condition is met. A loop statement allows programmers to execute a statement or group of statements multiple times without repetition of code.

```c
// C program to illustrate need of loops
#include <stdio.h>

int main()
{
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");

    return 0;
}
```

There are mainly two types of loops in C Programming:

1. **Entry Controlled loops:** In Entry controlled loops the test condition is checked before entering the main body of the loop. **For Loop and While Loop** is Entry-controlled loops.
2. **Exit Controlled loops:** In Exit controlled loops the test condition is evaluated at the end of the loop body. The loop body will execute at least once, irrespective of whether the condition is true or false. **do-while Loop** is Exit Controlled loop.



for Loop

for loop in C programming is a repetition control structure that allows programmers to write a loop that will be executed a specific number of times.

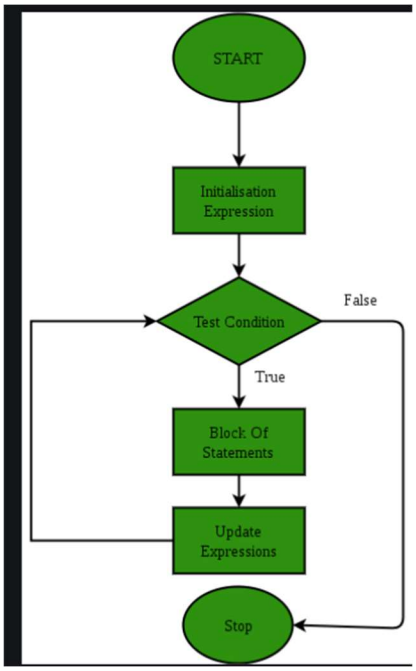
Syntax:

```
for (initialize expression; test expression; update expression)
{
    //
    // body of for loop
    //
}
```

Example:

```
for(int i = 0; i < n; ++i)
{
    printf("Body of for loop which will execute till n");
}
```

- **Initialization Expression:** In this expression, we assign a loop variable or loop counter to some value. for example: int i=1;
- **Test Expression:** In this expression, test conditions are performed. If the condition evaluates to true then the loop body will be executed and then an update of the loop variable is done. If the test expression becomes false then the control will exit from the loop. for example, i<=9;
- **Update Expression:** After execution of the loop body loop variable is updated by some value it could be incremented, decremented, multiplied, or divided by any value.



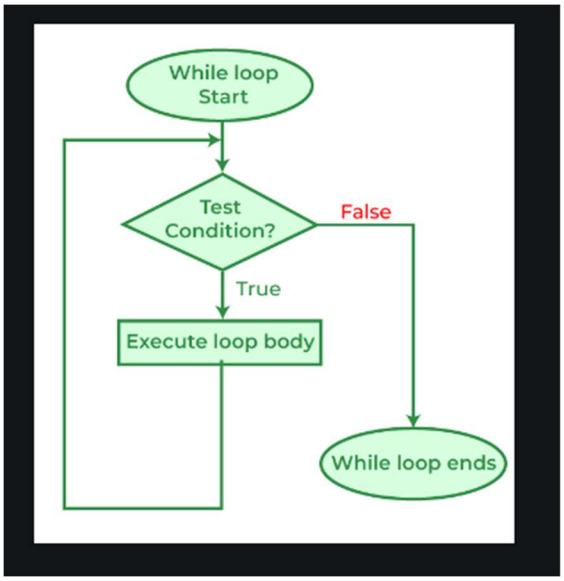
C

```c
// C program to illustrate for loop
#include <stdio.h>

// Driver code
int main()
{
  int i = 0;

  for (i = 1; i <= 10; i++)
  {
    printf( "Hello World\n");
  }
  return 0;
}
```

While Loop

While loop does not depend upon the number of iterations. In for loop the number of iterations was previously known to us but in the While loop, the execution is terminated on the basis of the test condition. If the test condition will become false then it will break from the while loop else body will be executed.

```
initialization_expression;

while (test_expression)
{
    // body of the while loop

    update_expression;
}
```



C

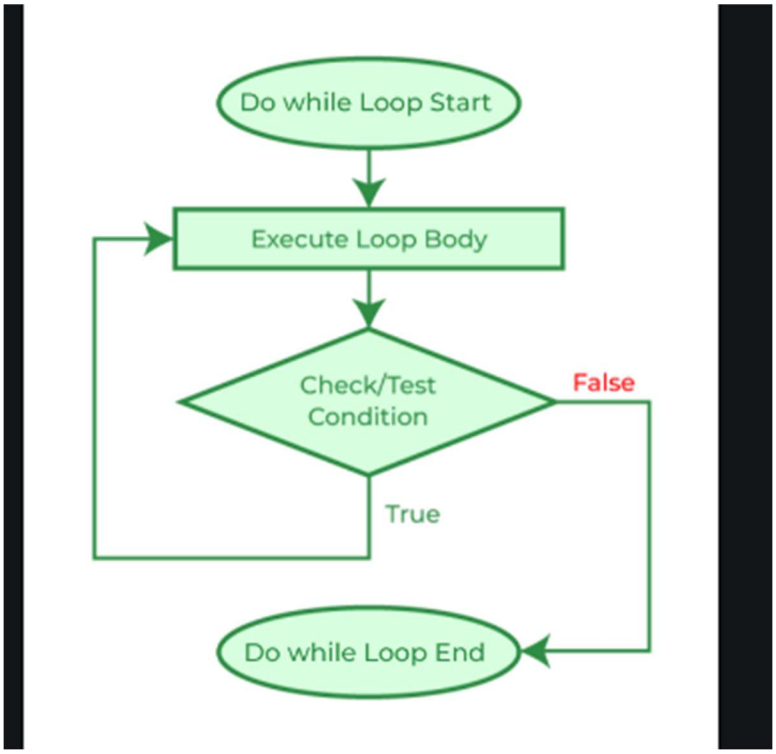
```c
// C program to illustrate
// while loop
#include <stdio.h>

// Driver code
int main()
{
  // Initialization expression
  int i = 2;

  // Test expression
  while(i < 10)
  {
    // loop body
    printf( "Hello World\n");

    // update expression
    i++;
  }

  return 0;
}
```

do-while Loop

The do-while loop is similar to a while loop but the only difference lies in the do-while loop test condition which is tested at the end of the body. In the do-while loop, the loop body will **execute at least once** irrespective of the test condition.

Syntax:

```
initialization_expression;
do
{
    // body of do-while loop


    update_expression;

} while (test_expression);
```



C

```c
// C program to illustrate
// do-while loop
#include <stdio.h>

// Driver code
int main()
{
  // Initialization expression
  int i = 2;

  do
  {
    // loop body
    printf( "Hello World\n");

    // Update expression
    i++;

    // Test expression
  }  while (i < 1);

  return 0;
}
```

Infinite Loop

An infinite loop is executed when the test expression never becomes false and the body of the loop is executed repeatedly. A program is stuck in an Infinite loop when the condition is always true. Mostly this is an error that can be resolved by using Loop Control statements.

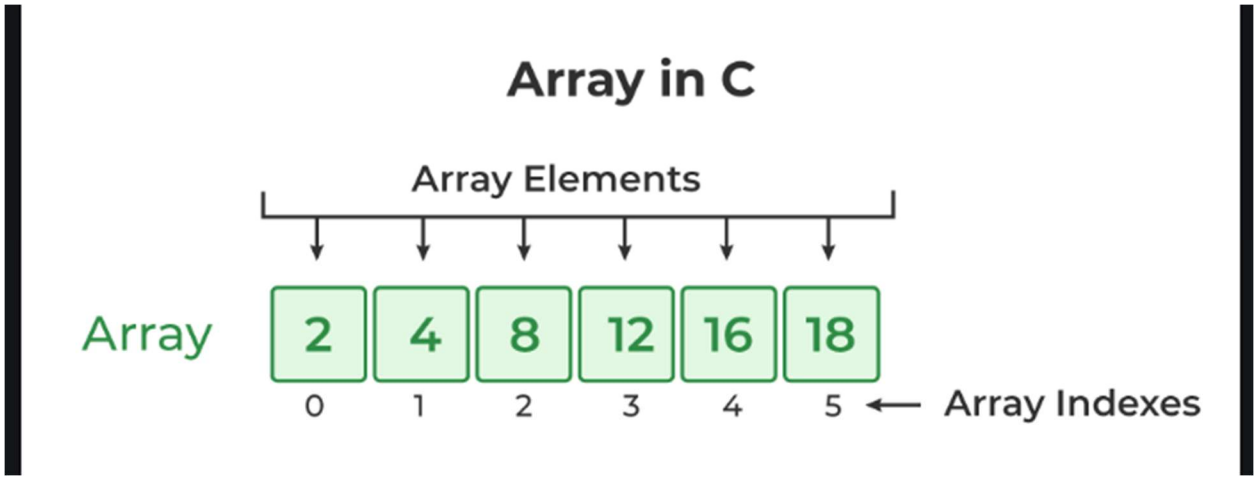
Using While loop:

C

```c
// C program to demonstrate
// infinite loop using while
// loop
#include <stdio.h>

// Driver code
int main()
{
  while (1)
    printf("This loop will run forever.\n");
  return 0;
}
```

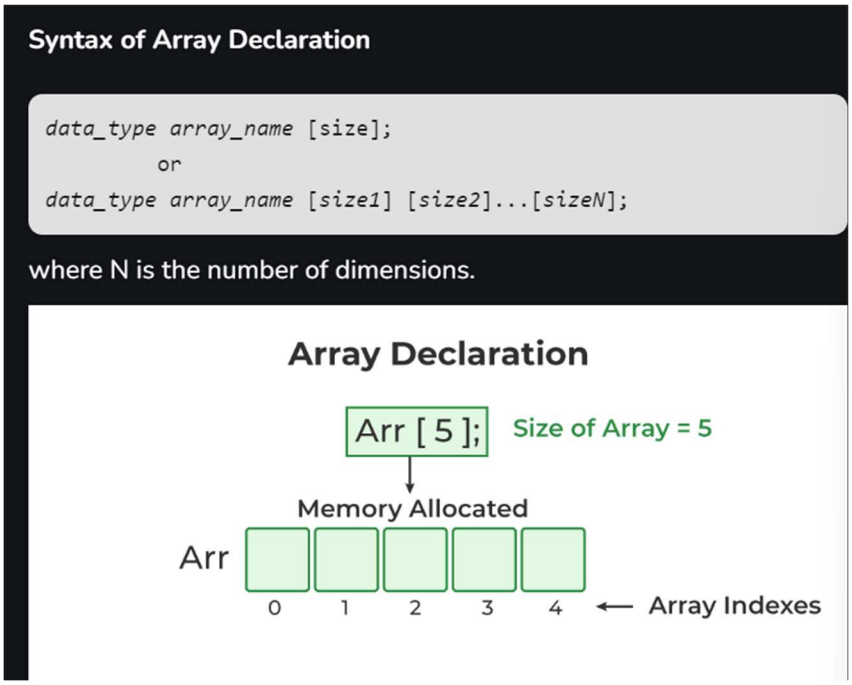
C Arrays

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.



C Array Declaration

In C, we have to declare the array like any other variable before using it. We can declare an array by specifying its name, the type of its elements, and the size of its dimensions. When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.



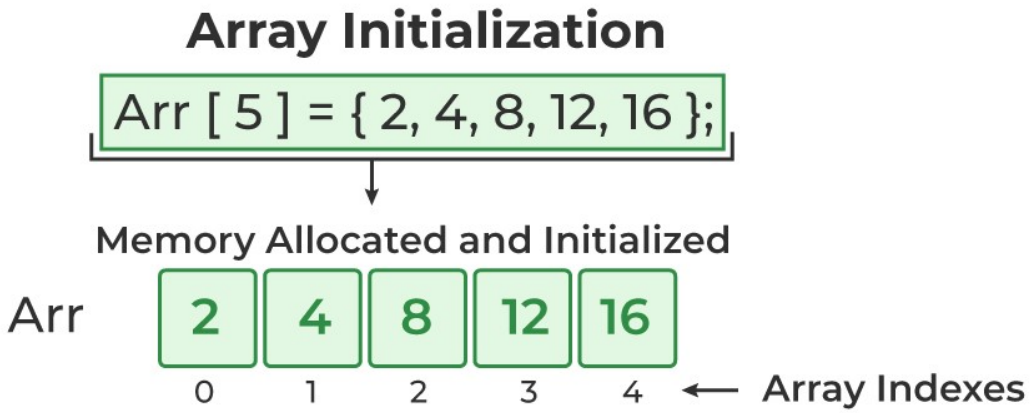
C Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful value. There are multiple ways in which we can initialize an array in C.

1. Array Initialization with Declaration

In this method, we initialize the array along with its declaration. We use an initializer list to initialize multiple elements of the array. An initializer list is the list of values enclosed within braces **{ }** separated by a comma.

data_type array_name [*size*] = {value1, value2, ... valueN};



2. Array Initialization with Declaration without Size

If we initialize an array using an initializer list, we can skip declaring the size of the array as the compiler can automatically deduce the size of the array in these cases. The size of the array in these cases is equal to the number of elements present in the initializer list as the compiler can automatically deduce the size of the array.

```
data_type array_name[] = {1,2,3,4,5};
```

The size of the above arrays is 5 which is automatically deduced by the compiler.

3. Array Initialization after Declaration (Using Loops)

We initialize the array after the declaration by assigning the initial value to each element individually. We can use for loop, while loop, or do-while loop to assign the value to each element of the array.

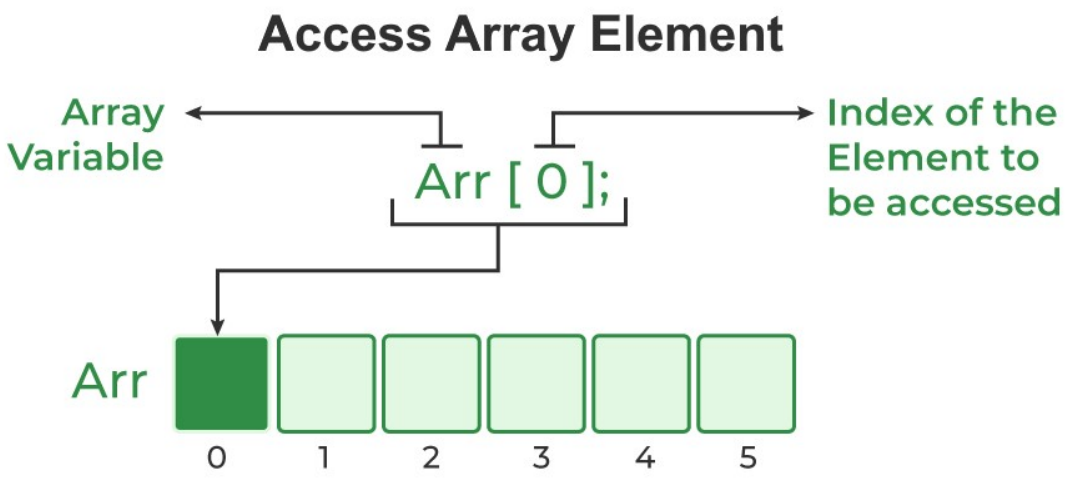
```
for (int i = 0; i < N; i++) {
    array_name[i] = valuei;
}
```

Access Array Elements

We can access any element of an array in C using the array subscript operator **[ ]** and the index value ***i*** of the element.

```
array_name [index];
```

One thing to note is that the indexing in the array always starts with 0, i.e., the **first element** is at index **0** and the **last element** is at **N – 1** where **N** is the number of elements in the array.



Update Array Element

We can update the value of an element at the given index i in a similar way to accessing an element by using the array subscript operator **[ ]** and assignment operator **=**.

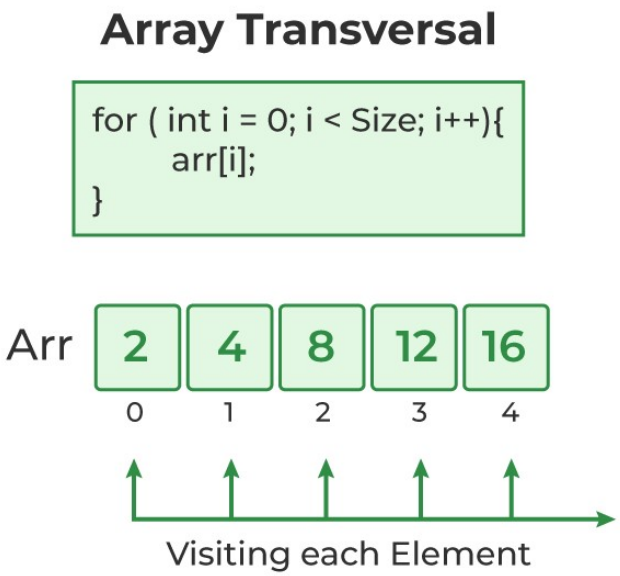
```
array_name[i] = new_value;
```

C Array Traversal

Traversal is the process in which we visit every element of the data structure. For C array traversal, we use loops to iterate through each element of the array.

Array Traversal using for Loop

```
for (int i = 0; i < N; i++) {
    array_name[i];
}
```



Types of Array in C

There are two types of arrays based on the number of dimensions it has. They are as follows:

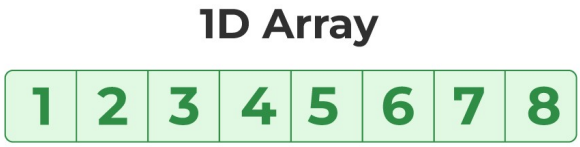
1. One Dimensional Arrays (1D Array)
2. Multidimensional Arrays

1. One Dimensional Array in C

The One-dimensional arrays, also known as 1-D arrays in C are those arrays that have only one dimension.

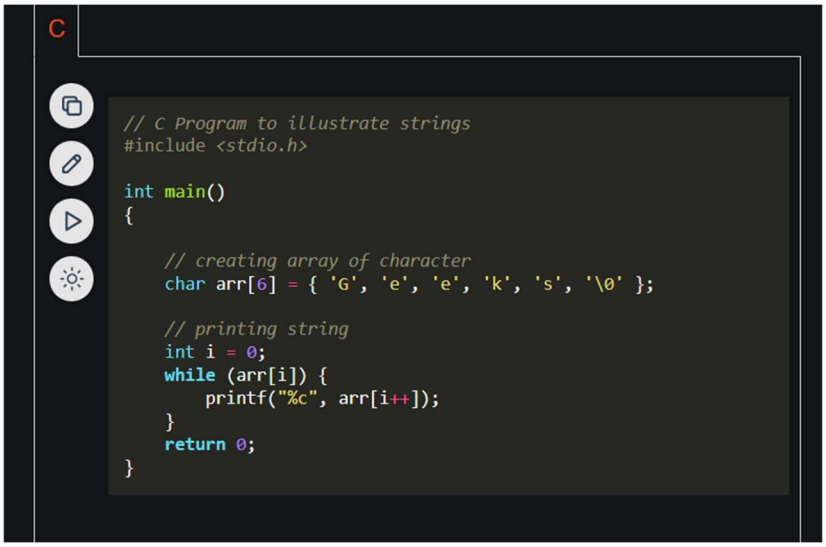
Syntax of 1D Array in C

array_name [size];



Array of Characters (Strings)

In C, we store the words, i.e., a sequence of characters in the form of an array of characters terminated by a NULL character. These are called strings in C language.



2. Multidimensional Array in C

Multi-dimensional Arrays in C are those arrays that have more than one dimension. Some of the popular multidimensional arrays are 2D arrays and 3D arrays. We can declare arrays with more dimensions than 3d arrays but they are avoided as they get very complex and occupy a large amount of space.

A. Two-Dimensional Array in C

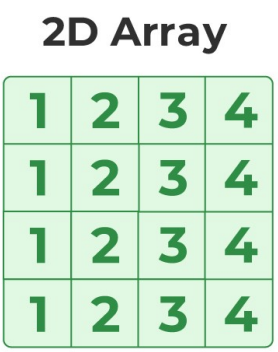
A Two-Dimensional array or 2D array in C is an array that has exactly two dimensions. They can be visualized in the form of rows and columns organized in a two-dimensional plane.

Syntax of 2D Array in C

array_name[size1] [size2];

Here,

- **size1:** Size of the first dimension.
- **size2:** Size of the second dimension.



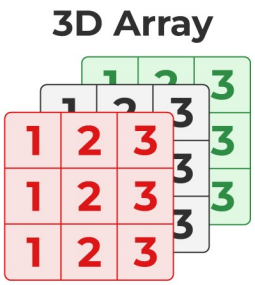

```c
// C Program to illustrate 2d array
#include <stdio.h>

int main()
{

    // declaring and initializing 2d array
    int arr[2][3] = { 10, 20, 30, 40, 50, 60 };

  printf("2D Array:\n");
    // printing 2d array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ",arr[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

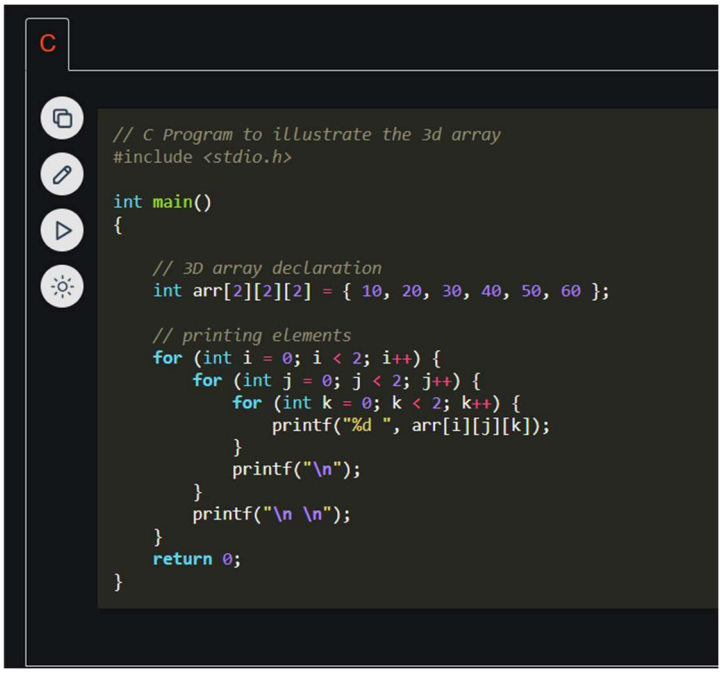
B. Three-Dimensional Array in C

Another popular form of a multi-dimensional array is Three Dimensional Array or 3D Array. A 3D array has exactly three dimensions. It can be visualized as a collection of 2D arrays stacked on top of each other to create the third dimension.

Syntax of 3D Array in C

array_name [size1] [size2] [size3];





Properties of Arrays in C

It is very important to understand the properties of the C array so that we can avoid bugs while using it. The following are the main properties of an array in C:

1. Fixed Size

The array in C is a fixed-size collection of elements. The size of the array must be known at the compile time and it cannot be changed once it is declared.

2. Homogeneous Collection

We can only store one type of element in an array. There is no restriction on the number of elements but the type of all of these elements must be the same.

3. Indexing in Array

The array index always starts with 0 in C language. It means that the index of the first element of the array will be 0 and the last element will be N – 1.

4. Dimensions of an Array

A dimension of an array is the number of indexes required to refer to an element in the array. It is the number of directions in which you can grow the array size.

5. Contiguous Storage

All the elements in the array are stored continuously one after another in the memory. It is one of the defining properties of the array in C which is also the reason why random access is possible in the array.

6. Random Access

The array in C provides random access to its element i.e we can get to a random element at any index of the array in constant time complexity just by using its index number.

7. No Index Out of Bounds Checking

There is no index out-of-bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

Advantages of Array in C

The following are the main advantages of an array:

1. Random and fast access of elements using the array index.
2. Use of fewer lines of code as it creates a single array of multiple elements.
3. Traversal through the array becomes easy using a single loop.
4. Sorting becomes easy as it can be accomplished by writing fewer lines of code.

Disadvantages of Array in C

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be rearranged after insertion and deletion.

Strings in C

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is stored as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character '\0'.

C String Declaration Syntax

```
char string_name[size];
```

In the above syntax **string_name** is any name given to the string variable and size is used to define the length of the string.

C String Initialization

We can initialize a C string in 4 different ways which are as follows:

1. Assigning a String Literal without Size

String literals can be assigned without size. Here, the name of the string str acts as a pointer because it is an array.

```
char str[] = "GeeksforGeeks";
```

2. Assigning a String Literal with a Predefined Size

String literals can be assigned with a predefined size. But we should always account for one extra space which will be assigned to the null character.

```
char str[50] = "GeeksforGeeks";
```

3. Assigning Character by Character with Size

We can also assign a string character by character. But we should remember to set the end character as '\0' which is a null character.

```
char str[14] = { 'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

4. Assigning Character by Character without Size

We can assign character by character without size with the NULL character at the end. The size of the string is determined by the compiler automatically.

```
char str[] = { 'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

Note: *When a Sequence of characters enclosed in the double quotation marks is encountered by the compiler, a null character '\0' is appended at the end of the string by default.*

Read a String Input From the User

The following example demonstrates how to take string input using scanf() function in C

C

```c
// C program to read string from user
#include<stdio.h>

int main()
{
    // declaring string
    char str[50];

    // reading string
    scanf("%s",str);

    // print string
    printf("%s",str);

    return 0;
}
```

Standard C Library – String.h Functions

The C language comes bundled with <string.h> which contains some useful string-handling functions. Some of them are as follows:

Function Name	Description
strlen(string_name)	Returns the length of string name.
strcpy(s1, s2)	Copies the contents of string s2 to string s1.
strcmp(str1, str2)	Compares the first string with the second string. If strings are the same it returns 0.
strcat(s1, s2)	Concat s1 string with s2 string and the result is stored in the first string.
strlwr()	Converts string to lowercase.
strupr()	Converts string to uppercase.
strstr(s1, s2)	Find the first occurrence of s2 in s1.

Array of Strings in C

In C programming String is a 1-D array of characters and is defined as an array of characters. But an array of strings in C is a two-dimensional array of character types. Each String is terminated with a null character (\0). It is an application of a 2d array.

Syntax:

char variable_name[r][m] = {list of string};

Here,

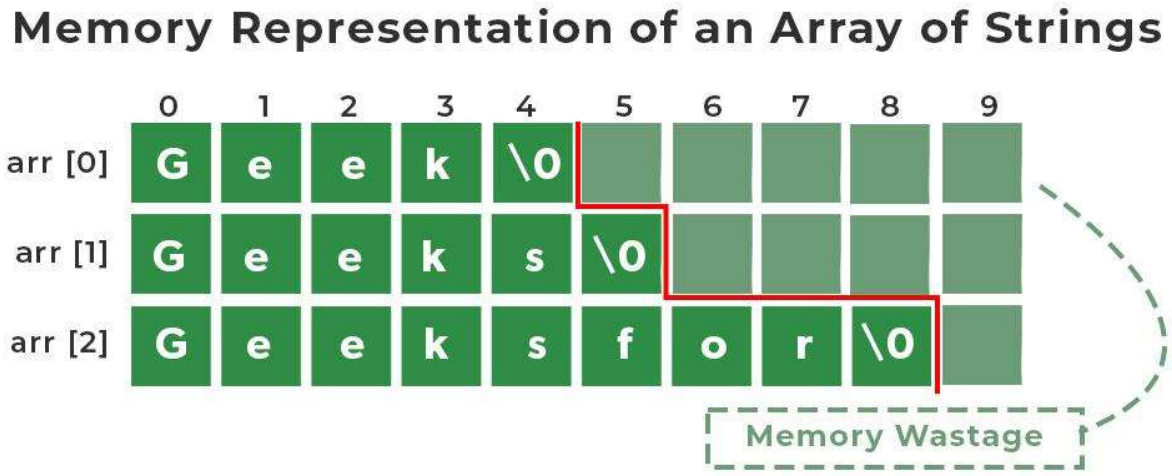
- **var_name** is the name of the variable in C.
- **r** is the maximum number of string values that can be stored in a string array.
- **m** is the maximum number of character values that can be stored in each string array.

Example:

C

```c
// C Program to print Array
// of strings
#include <stdio.h>

// Driver code
int main()
{
  char arr[3][10] = {"Geek",
                     "Geeks", "Geekfor"};
  printf("String array Elements are:\n");

  for (int i = 0; i < 3; i++)
  {
    printf("%s\n", arr[i]);
  }
  return 0;
}
```

Below is the Representation of the above program



We have 3 rows and 10 columns specified in our Array of String but because of prespecifying, the size of the array of strings the space consumption is high. So, to avoid high space consumption in our program we can use an Array of Pointers in C.

Invalid Operations in Arrays of Strings

We can't directly change or assign the values to an array of strings in C.

```
char arr[3][10] = {"Geek", "Geeks", "Geekfor"};
arr[0] = "GFG";  // This will give an Error that says
assignment to expression with an array type.
```

To change values we can use strcpy() function in C

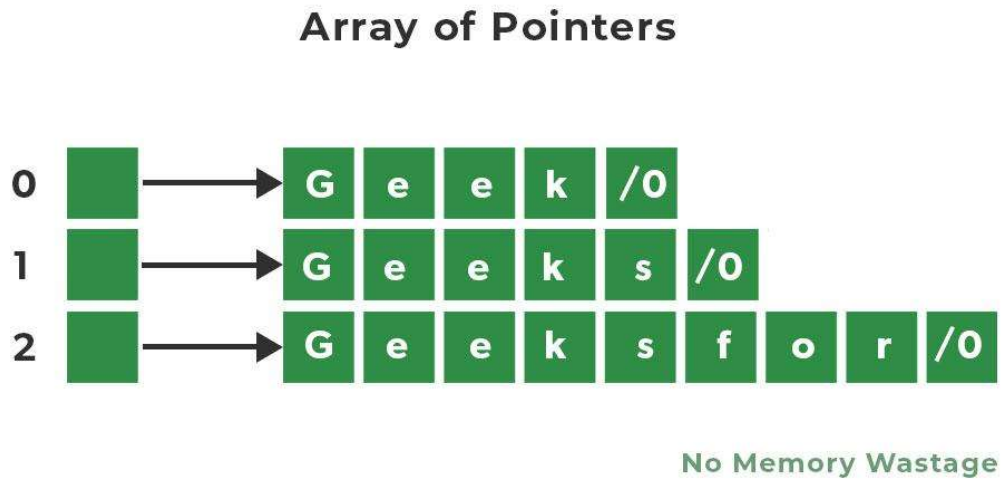
```
strcpy(arr[0],"GFG"); // This will copy the value to the
arr[0].
```

Array of Pointers of Strings

In C we can use an Array of pointers. Instead of having a 2-Dimensional character array, we can have a single-dimensional array of Pointers. Here pointer to the first character of the string literal is stored.

Syntax:

char *arr[] = { "Geek", "Geeks", "Geekfor" };




```c
// C Program to print Array
// of Pointers
#include <stdio.h>

// Driver code
int main()
{
  char *arr[] = {"Geek", "Geeks", "Geekfor"};
  printf("String array Elements are:\n");

  for (int i = 0; i < 3; i++)
  {
    printf("%s\n", arr[i]);
  }
  return 0;
}
```

C String Functions

The C string functions are built-in functions that can be used for various operations and manipulations on strings. These string functions can be used to perform tasks such as string copy, concatenation, comparison, length, etc. The **<string.h>** header file contains these string functions.

1. strcat() Function

The strcat() function in C is used for string concatenation. It will append a copy of the source string to the end of the destination string.

Syntax

char* **strcat**(char* *dest*, const char* *src*);

The terminating character at the end of **dest** is replaced by the first character of **src.**

Parameters

- **dest**: Destination string
- **src**: Source string

Return value

- The strcat() function returns a pointer to the dest string.

Example

```
// C Program to illustrate the strcat function
#include <stdio.h>

int main()
{
    char dest[50] = "This is an";
    char src[50] = " example";

    printf("dest Before: %s\n", dest);

    // concatenating src at the end of dest
```

```
    strcat(dest, src);


    printf("dest After: %s", dest);


    return 0;
}
```

Output

dest Before: This is an

dest After: This is an example

In C, there is a function strncat() similar to strcat().

strncat()

This function is used for string handling. This function appends not more than n characters from the string pointed to by **src** to the end of the string pointed to by **dest** plus a terminating Null-character.

Syntax of strncat()

char* **strncat**(char* *dest*, const char* *src*, size_t *n*);

where **n** represents the maximum number of characters to be appended. size_t is an unsigned integral type.

2. strlen() Function

The strlen() function calculates the length of a given string. It doesn't count the null character '\0'.

Syntax

int **strlen**(const char **str*);

Parameters

- **str:** It represents the string variable whose length we have to find.

Return Value

- strlen() function in C returns the length of the string.

Example

C

```c
// C program to demonstrate the strlen() function
#include <stdio.h>
#include <string.h>

int main()
{
    // Declare and initialize a character array 'str' with
    // the string "GeeksforGeeks"
    char str[] = "GeeksforGeeks";

    // Calculate the length of the string using the strlen()
    // function and store it in the variable 'length'
    size_t length = strlen(str);

    // Print the length of the string
    printf("String: %s\n", str);

    printf("Length: %zu\n", length);

    return 0;
}
```

Output

String: GeeksforGeeks

Length: 13

3. strcmp() Function

The strcmp() is a built-in library function in C. This function takes two strings as arguments and compares these two strings lexicographically.

Syntax

int **strcmp**(const char **str1*, const char **str2*);

Parameters

- **str1**: This is the first string to be compared.
- **str2**: This is the second string to be compared.

Return Value

- If str1 is less than str2, the return value is less than 0.
- If str1 is greater than str2, the return value is greater than 0.
- If str1 is equal to str2, the return value is 0.

Example

C

```
// C program to demonstrate the strcmp() function
#include <stdio.h>
#include <string.h>

int main()
{
    // Define a string 'str1' and initialize it with "Geeks"
    char str1[] = "Geeks";
    // Define a string 'str2' and initialize it with "For"
    char str2[] = "For";
    // Define a string 'str3' and initialize it with "Geeks"
    char str3[] = "Geeks";
```

```
    // Compare 'str1' and 'str2' using strcmp() function and
    // store the result in 'result1'
    int result1 = strcmp(str1, str2);
    // Compare 'str2' and 'str3' using strcmp() function and
    // store the result in 'result2'
    int result2 = strcmp(str2, str3);
    // Compare 'str1' and 'str1' using strcmp() function and
    // store the result in 'result3'
    int result3 = strcmp(str1, str1);

    // Print the result of the comparison between 'str1' and
    // 'str2'
    printf("Comparison of str1 and str2: %d\n", result1);
    // Print the result of the comparison between 'str2' and
    // 'str3'
    printf("Comparison of str2 and str3: %d\n", result2);
    // Print the result of the comparison between 'str1' and
    // 'str1'
    printf("Comparison of str1 and str1: %d\n", result3);

    return 0;
}
```

Output

```
Comparison of str1 and str2: 1
Comparison of str2 and str3: -1
Comparison of str1 and str1: 0
```

There is a function **strncmp()** similar to strcmp().

strncmp()

This function lexicographically compares the first n characters from the two null-terminated strings and returns an integer based on the outcome.

Syntax

int **strncmp**(const char* *str1*, const char* *str2*, size_t n*um*);

Where **num** is the number of characters to compare.

4. strcpy

The strcpy() is a standard library function in C and is used to copy one string to another. In C, it is present in **<string.h>** header file.

Syntax

char* **strcpy**(char* *dest*, const char* *src*);

Parameters

- **dest**: Pointer to the destination array where the content is to be copied.
- **src:** string which will be copied.

Return Value

- strcpy() function returns a pointer pointing to the output string.

Example

C

```
// C program to illustrate the use of strcpy()
#include <stdio.h>
#include <string.h>

int main()
{
   // defining strings

   char source[] = "GeeksforGeeks";
```

```
    char dest[20];


    // Copying the source string to dest
    strcpy(dest, source);


    // printing result
    printf("Source: %s\n", source);
    printf("Destination: %s\n", dest);


    return 0;
}
```

Output

Source: GeeksforGeeks

Destination: GeeksforGeeks

strncpy()

The function **strncpy()** is similar to strcpy() function, except that at most n bytes of src are copied.

If there is no NULL character among the first n character of src, the string placed in dest will not be NULL-terminated. If the length of src is less than n, strncpy() writes an additional NULL character to dest to ensure that a total of n characters are written.

Syntax

char* **strncpy**(char* *dest*, const char* *src*, size_t *n*);

Where n is the first n characters to be copied from src to dest.

5. strchr()

The strchr() function in C is a predefined function used for string handling. This function is used to find the first occurrence of a character in a string.

Syntax

char ***strchr**(const char **str*, int *c*);

Parameters

- **str:** specifies the pointer to the null-terminated string to be searched in.
- **ch:** specifies the character to be searched for.

Here, **str** is the string and **ch** is the character to be located. It is passed as its int promotion, but it is internally converted back to char.

Return Value

- It returns a pointer to the first occurrence of the character in the string.

Example

C++

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

int main()
{
    char str[] = "GeeksforGeeks";
    char ch = 'e';

    // Search for the character 'e' in the string
    // Use the strchr function to find the first occurrence
    // of 'e' in the string
    char* result = strchr(str, ch);

    // Character 'e' is found, calculate the index by
    // subtracting the result pointer from the str pointer
    if (result != NULL) {
        printf("The character '%c' is found at index %ld\n",
               ch, result - str);
    }
    else {
```

```
        printf("The character '%c' is not found in the "
               "string\n",
               ch);
    }

    return 0;
}
```

Output

The character 'e' is found at index 1

strrchr() Function

In C, strrchr() function is similar to strchr() function. This function is used to find the last occurrence of a character in a string.

Syntax

char* **strchr**(const char **str*, int *ch*);

6. strstr()

The strstr() function in C is used to search the first occurrence of a substring in another string.

Syntax

char ***strstr** (const char **s1*, const char **s2*);

Parameters

- **s1**: This is the main string to be examined.
- **s2**: This is the sub-string to be searched in the s1 string.

Return Value

- If the s2 is found in s1, this function returns a pointer to the first character of the s2 in s1, otherwise, it returns a null pointer.
- It returns s1 if s2 points to an empty string.

Example

C

```c
// C program to demonstrate the strstr() function
#include <stdio.h>
#include <string.h>

int main()
{
    // Define a string 's1' and initialize it with
    // "GeeksforGeeks"
    char s1[] = "GeeksforGeeks";
    // Define a string 's2' and initialize it with "for"
    char s2[] = "for";
    // Declare a pointer 'result' to store the result of
    // strstr()
    char* result;

    // Find the first occurrence of 's2' within 's1' using
    // strstr() function and assign the result to 'result'
    result = strstr(s1, s2);

    if (result != NULL) {
        // If 'result' is not NULL, it means the substring
        // was found, so print it
        printf("Substring found: %s\n", result);
    }
    else {
        // If 'result' is NULL, it means the substring was
        // not found, so print appropriate message
```

```
        printf("Substring not found.\n");
    }


    return 0;
}
```

Output

Substring found: forGeeks

7. strtok()

The strtok() function is used to split the string into small tokens based on a set of delimiter characters.

Syntax

char * **strtok**(char* *str*, const char **delims*);

Parameters

- **str:** It is the string to be tokenized.
- **delims:** It is the set of delimiters

Example

C++

```cpp
// C program to demonstrate the strtok() function
#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = "Geeks,for.Geeks";
    // Delimiters: space, comma, dot,
    // exclamation mark
    const char delimiters[] = ",.";

    // Tokenize the string
    char* token = strtok(str, delimiters);
    while (token != NULL) {
        printf("Token: %s\n", token);
        token = strtok(NULL, delimiters);
    }

    return 0;
}
```

Output

Token: Geeks

Token: for

Token: Geeks

Array of Pointers in C

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Note: *It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing. For example, enclosing *array_name in the parenthesis will mean that array_name is a pointer to an array.*

Example:

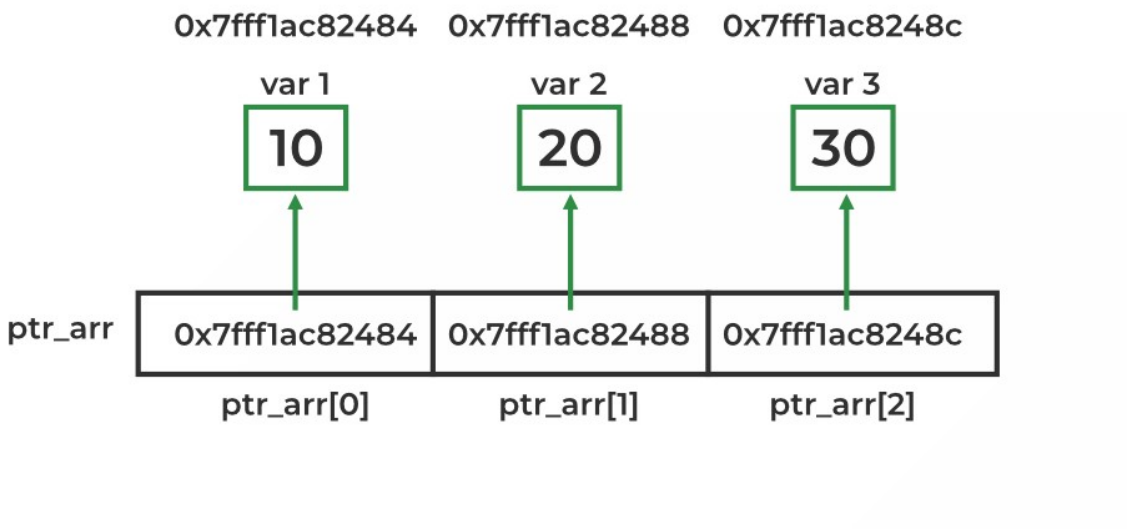
C

```c
// C program to demonstrate the use of array of pointers
#include <stdio.h>

int main()
{
    // declaring some temp variables
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;

    // array of pointers to integers
    int* ptr_arr[3] = { &var1, &var2, &var3 };

    // traversing using loop
    for (int i = 0; i < 3; i++) {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1,
*ptr_arr[i], ptr_arr[i]);
    }

    return 0;
}
```

Output

```
Value of var1: 10    Address: 0x7fff1ac82484
Value of var2: 20    Address: 0x7fff1ac82488
Value of var3: 30    Address: 0x7fff1ac8248c
```

Explanation:



As shown in the above example, each element of the array is a pointer pointing to an integer. We can access the value of these integers by first selecting the array element and then dereferencing it to get the value.

Array of Pointers to Character

One of the main applications of the array of pointers is to store multiple strings as an array of pointers to characters. Here, each pointer in the array is a character pointer that points to the first character of the string.

Syntax:

```
char *array_name [array_size];
```

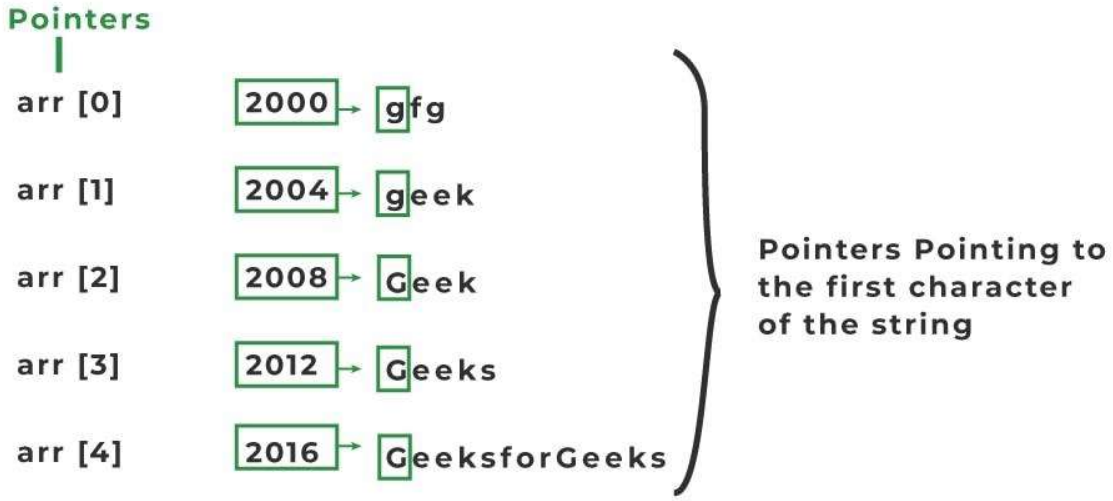
After that, we can assign a string of any length to these pointers.

Example:

Example:

C

```
char* arr[5]
    = { "gfg", "geek", "Geek", "Geeks", "GeeksforGeeks" }
```



Note: *Here, each string will take different amount of space so offset will not be the same and does not follow any particular order.*

This method of storing strings has the advantage of the traditional array of strings. Consider the following two examples:

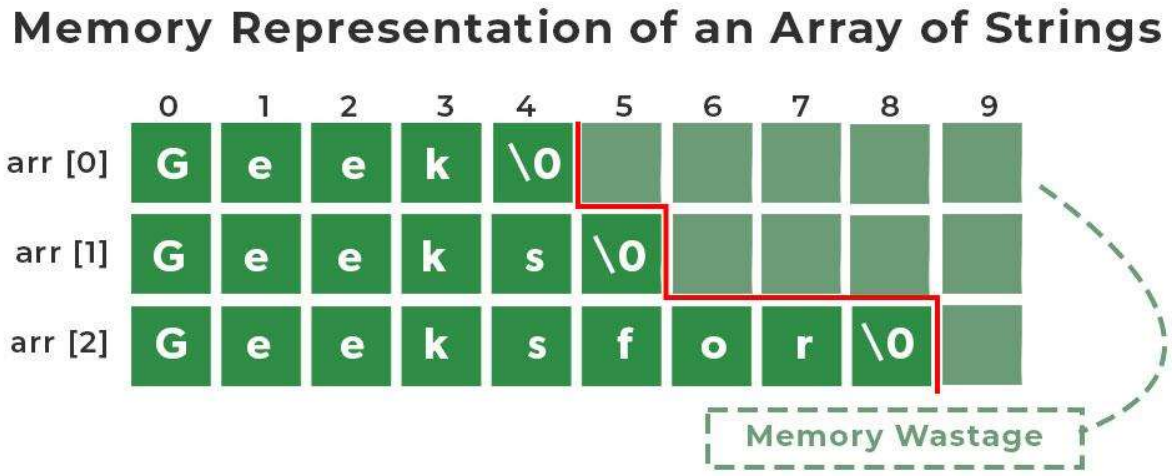
Example 1:

C

```c
// C Program to print Array of strings without array of
pointers
#include <stdio.h>
int main()
{
    char str[3][10] = { "Geek", "Geeks", "Geekfor" };

    printf("String array Elements are:\n");

    for (int i = 0; i < 3; i++) {
        printf("%s\n", str[i]);
    }

    return 0;
}
```

Output

```
String array Elements are:
Geek
Geeks
Geekfor
```



The space occupied by the array of pointers to characters is shown by solid green blocks excluding the memory required for storing the pointer while the space occupied by the array of strings includes both solid and light green blocks.

Array of Pointers to Different Types

Not only we can define the array of pointers for basic data types like int, char, float, etc. but we can also define them for derived and user-defined data types such as arrays, structures, etc. Let's consider the below example where we create an array of pointers pointing to a function for performing the different operations.

Example:

Application of Array of Pointers

An array of pointers is useful in a wide range of cases. Some of these applications are listed below:

- It is most commonly used to store multiple strings.
- It is also used to implement LinkedHashMap in C and also in the Chaining technique of collision resolving in Hashing.
- It is used in sorting algorithms like bucket sort.
- It can be used with any pointer type so it is useful when we have separate declarations of multiple entities and we want to store them in a single place.

Disadvantages of Array of Pointers

The array of pointers also has its fair share of disadvantages and should be used when the advantages outweigh the disadvantages. Some of the disadvantages of the array of pointers are:

- **Higher Memory Consumption:** An array of pointers requires more memory as compared to plain arrays because of the additional space required to store pointers.
- **Complexity:** An array of pointers might be complex to use as compared to a simple array.
- **Prone to Bugs:** As we use pointers, all the bugs associated with pointers come with it so we need to handle them carefully.

What is a Pointer in C?

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

As the pointers in C store the memory addresses, their size is independent of the type of data they are pointing to. This size of pointers in C only depends on the system architecture.

Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the **(*) dereferencing operator** in the pointer declaration.

```
datatype * ptr;
```

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. **Pointer Declaration**
2. **Pointer Initialization**
3. **Pointer Dereferencing**

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the **(*) dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the **(&: ampersand) addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Example

```
int var = 10;
int * ptr;
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

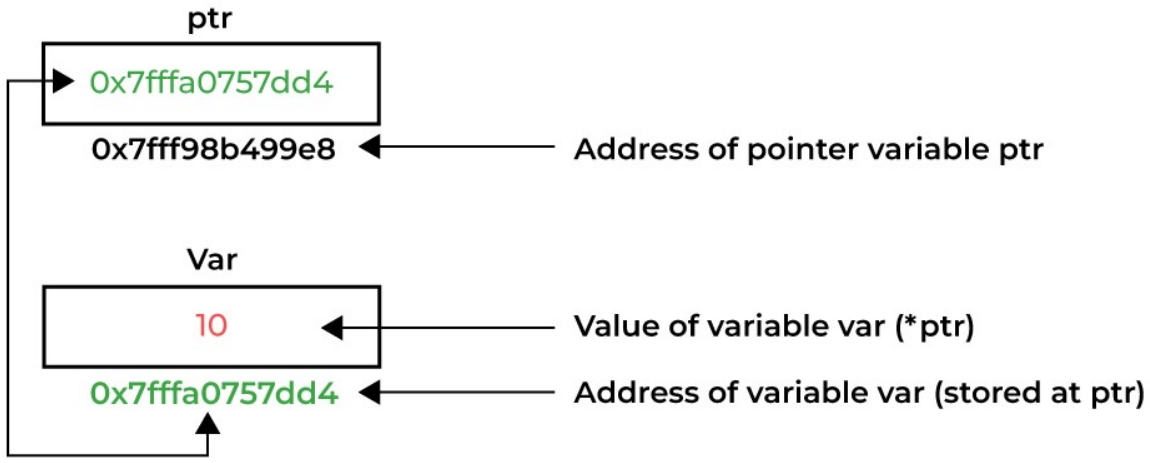
Example

```
int *ptr = &var;
```

Note: *It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.*

3. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **(*) dereferencing operator** that we used in the pointer declaration.



Dereferencing a Pointer in C

C

```c
// C program to illustrate Pointers
#include <stdio.h>

void geeks()
{
    int var = 10;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
}

// Driver program
int main()
{
    geeks();
    return 0;
}
```

Output

```
Value at ptr = 0x7ffca84068dc
Value at var = 10
Value at *ptr = 10
```

Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax

```
int *ptr;
```

These pointers are pronounced as **Pointer to Integer.**

Similarly, a pointer can point to any primitive data type. It can point also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an array using the given syntax.

Syntax

```
char *ptr = &array_name;
```

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax

```
struct struct_name *ptr;
```

In C, structure pointers are used in data structures such as linked lists, trees, etc.

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the function pointer for this function will be

Syntax

```
int (*ptr)(int, char);
```

Note: The syntax of the function pointers changes according to the function prototype.

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

Syntax

datatype ** pointer_name;

Dereferencing Double Pointer

*pointer_name; // get the address stored in the inner level pointer
**pointer_name; // get the value pointed by inner level pointer

Note: *In C, we can create multi-level pointers with any number of levels such as – ***ptr3, ****ptr4, ******ptr5 and so on.*

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax

data_type **pointer_name* = NULL;
or
pointer_name = NULL

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The Void pointers in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

Syntax

void * *pointer_name*;

One of the main properties of void pointers is that they cannot be dereferenced.

8. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values is updated using wild pointers, they could cause data abort or data corruption.

Example

```
int *ptr;
char *str;
```

9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax

data_type * const *pointer_name*;

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax

const *data_type* * *pointer_name*;

Other Types of Pointers in C:

There are also the following types of pointers available to use in C apart from those specified above:

- **Far pointer:** A far pointer is typically 32-bit that can access memory outside the current segment.
- **Dangling pointer:** A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
- **Huge pointer:** A huge pointer is 32-bit long containing segment address and offset address.
- **Complex pointer:** Pointers with multiple levels of indirection.
- **Near pointer:** Near pointer is used to store 16-bit addresses means within the current segment on a 16-bit machine.
- **Normalized pointer:** It is a 32-bit pointer, which has as much of its value in the segment register as possible.
- **File Pointer:** The pointer to a FILE data type is called a stream pointer or a file pointer.

Size of Pointers in C

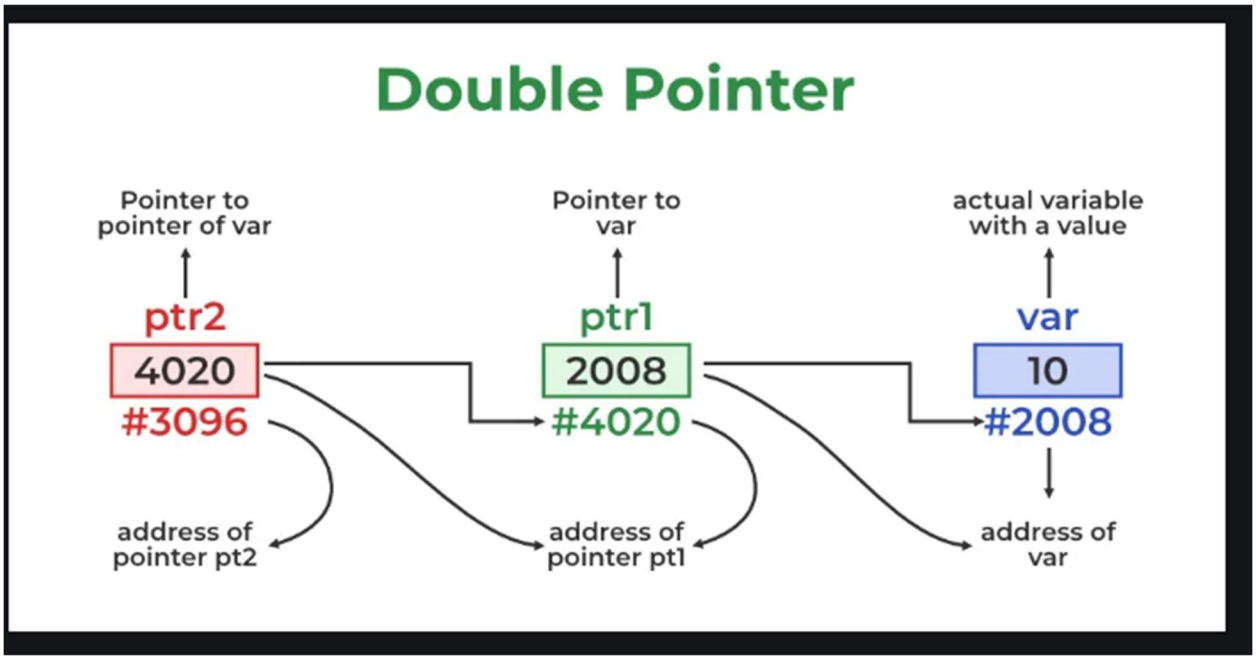
The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

C – Pointer to Pointer (Double Pointer)

The pointer to a pointer in C is used when we want to store the address of another pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as ***double-pointers***. We can use a pointer to a pointer to change the values of normal pointers or create a variable-sized 2-D array. A double pointer occupies the same amount of space in the memory stack as a normal pointer.



Declaration of Pointer to a Pointer in C

Declaring Pointer to Pointer is similar to declaring a pointer in C. The difference is we have to place an additional '*' before the name of the pointer.

```
data_type_of_pointer **name_of_variable = &
normal_pointer_variable;
```

```
int val = 5;
int *ptr = &val;    // storing address of val to pointer ptr.
int **d_ptr = &ptr; // pointer to a pointer declared
                    // which is pointing to an integer.
```

The above diagram shows the memory representation of a pointer to a pointer. The first pointer ptr1 stores the address of the variable and the second pointer ptr2 stores the address of the first pointer.

Example of Double Pointer in C

C

```
// C program to demonstrate pointer to pointer
#include <stdio.h>

int main()
{
    int var = 789;

    // pointer for var
    int* ptr2;

    // double pointer for ptr2
    int** ptr1;
```

```
    // storing address of var in ptr2
    ptr2 = &var;

    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;

    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var);
    printf("Value of var using single pointer = %d\n", *ptr2);
    printf("Value of var using double pointer = %d\n", **ptr1);

    return 0;
}
```

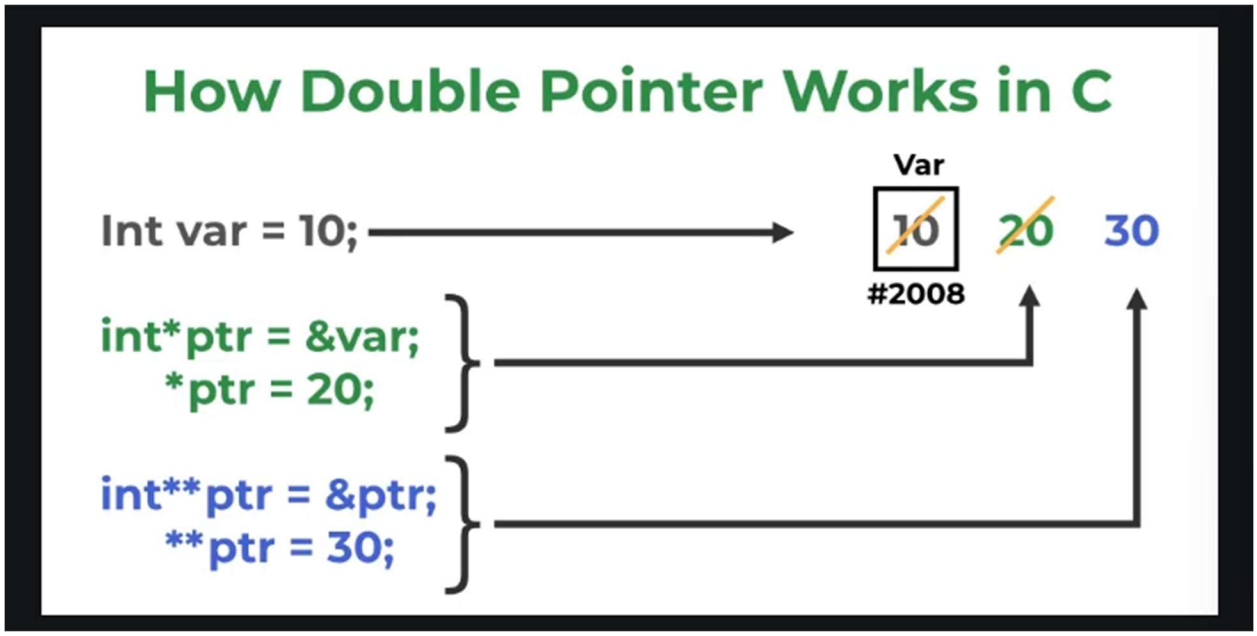
Output

Value of var = 789

Value of var using single pointer = 789

Value of var using double pointer = 789

How Double Pointer Works?



The working of the double-pointer can be explained using the above image:

- The double pointer is declared using the syntax shown above.
- After that, we store the address of another pointer as the value of this new double pointer.
- Now, if we want to manipulate or dereference to any of its levels, we have to use Asterisk (*) operator the number of times down the level we want to go.

User-Defined Data Types In C

Data Types are the types of data that can be stored in memory using a programming language. Basically, data types are used to indicate the type of data that a variable can store. These data types require different amounts of memory and there are particular operations that can be performed on them. These data types can be broadly classified into three types:

1. Primitive Data Types
2. Derived Types
3. User Defined Data Types

In this article, will discuss the User-Defined Data Type.

What is a user-defined Datatype in C?

The data types defined by the user themself are referred to as user-defined data types. These data types are derived from the existing datatypes.

Need of User-Defined Datatypes

- It enables customization in the code.
- Users can write more efficient and flexible code.
- Provides abstraction.

Types of User-Defined DataTypes

There are 4 types of user-defined data types in C. They are

1. Structure
2. Union
3. Enum
4. Typedef

1. Structure

As we know, C doesn't have built-in object-oriented features like C++ but structures can be used to achieve encapsulation to some level. Structures are used to group items of different types into a single type. The "struct" keyword is used to define a structure. The size of the structure is equal to or greater than the total size of all of its members.

Syntax

struct *structure_name* {

data_type *member_name1;*

```
    data_type member_name1;
    ....
    ....
};
```

Example

C

```c
// C Code to implement a struct
#include <stdio.h>

// Defining a structure
struct Person {
    char company[50];
    int lifespan;
};

int main()
{
    // Declaring a variable of the structure type
    struct Person person1;

    // Initializing structure members
    strcpy(person1.company, "GeeksforGeeks");
    person1.lifespan = 30;

    // Accessing and printing structure members
    printf("Name: %s\n", person1.company);
    printf("Age: %d\n", person1.lifespan);
```

```
    return 0;
}
```

Output

Name: GeeksforGeeks

Age: 30

2. Union

Unions are similar to structures in many ways. What makes a union different is that all the members in the union are stored in the same memory location resulting in only one member containing data at the same time. The size of the union is the size of its largest member. Union is declared using the “union” keyword.

Syntax

union *union_name* {

   datatype *member1;*

   datatype *member2;*

   ...

};

Example

C

```
// C Code to implement Union
#include <stdio.h>


// Declaring a union
union Data {
   int i;
   float f;
   char str[20];
};
```

```
int main()
{
    // creating an instance named 'data' of union Data
    union Data data;
    data.i = 10;
    printf("Data.i: %d\n", data.i);

    data.f = 3.14;
    printf("Data.f: %f\n", data.f);

    strcpy(data.str, "GeeksforGeeks, C");
    printf("Data.str: %s\n", data.str);
    return 0;
}
```

Output

Data.i: 10

Data.f: 3.140000

Data.str: GeeksforGeeks, C

2. Enumeration (enums)

Enum is short for “Enumeration”. It allows the user to create custom data types with a set of named integer constants. The “enum” keyword is used to declare an enumeration. Enum simplifies and makes the program more readable.

Syntax

enum *enum_name* {const1, const2, ..., constN};

Here, the const1 will be assigned 0, const2 = 1, and so on in the sequence.

We can also assign a custom integer value such as:

enum *enum_name* {

```
    const1 = 8;
    const2 = 4;
}
```

Example

C

```
// C code to implement enum
#include <stdio.h>

// Declaring a enum
enum Cafes {
    Dyu_Art_Cafe,
    Tea_Villa_Cafe,
    The_Hole_in_the_Wall_Cafe,
    Cafe_Azzure,
    The_Banaglore_Cafe,
    Dialogues_Cafe,
    Cafe_Coffee_Day
};

// driver code
int main()
{
    enum Cafes today = The_Banaglore_Cafe;
    printf("Today is %d\n", today);
    return 0;
}
```

Output

Today is 4

Typedef

typedef is used to redefine the existing data type names. Basically, it is used to provide new names to the existing data types. The “typedef” keyword is used for this purpose;

Syntax

typedef *existing_name alias_name;*

Example

C

```
// C Code to implement Typedef
#include <stdio.h>

typedef char Company;

int main()
{
    Company* name = "GeeksforGeeks";
    printf("Best Tutorials on: %s \n", name);
    return 0;
}
```

Output

Best Tutorials on: GeeksforGeeks