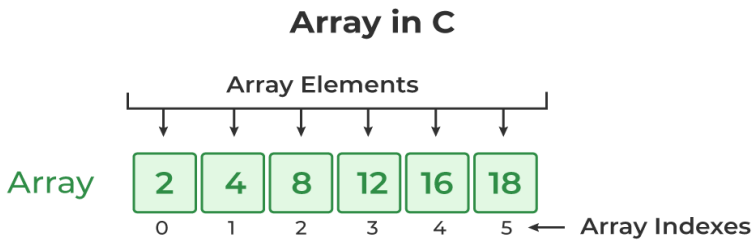
Unit - 4 : Arrays & Strings, Structure & Union

Arrays

Define Array: An Array is a fixed –size sequenced collection of elements of the same data type.It can be used to reperesent a list of numbers or list of names. 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.

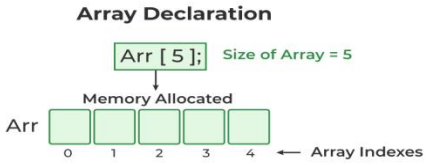


C Array Declaration

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.

Syntax of Array Declaration

```
data_type array_name [size];
      or
data_type array_name [size1] [size2]...[sizeN];
```



Example:

```
// C Program to illustrate the array declaration
#include <stdio.h>
int main()
{
// declaring array of integers
int arr_int[5];
// declaring array of characters
char arr_char[5];
return 0;
}
```

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 initialize list is the list of values enclosed within braces { } separated b 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};
```

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;
}
```

Example of Array Initialization in C

```
// C Program to demonstrate array initialization
#include <stdio.h>
int main()
{
// array initialization using initialize list
int arr[5] = { 10, 20, 30, 40, 50 };
// array initialization using initializer list without
// specifying size
int arr1[] = { 1, 2, 3, 4, 5 };
// array initialization using for loop
float arr2[5];
for (int i = 0; i < 5; i++)
{
arr2[i] = (float)i * 2.1;
}
return 0;
}
```

Types of Arrays

There are majorly three types of arrays:

1. One-dimensional array (1-D arrays)
2. Two-dimensional (2D) array
3. Three-dimensional array

One dimensional array in C

Arrays are a fundamental concept in programming, and they come in different dimensions. One-dimensional arrays, also known as single arrays, are arrays with only one dimension or a single row.

Syntax of One-Dimensional Array in C

dataType arrayName [arraySize];

dataType : specifies the data type of the array. It can be any valid data type in C programming language, such as int, float, char, double, etc.

arrayName: is the name of the array, which is used to refer to the array in the program.

arraySize: specifies the number of elements in the array. It must be a positive integer value.

Example of One-Dimensional Array in C

```
#include <stdio.h>
 int main()
{
int  numbers[5] = {10, 20, 30, 40, 50};
for(int i=0; i<5; i++)
{
printf("numbers[%d] = %d\n", i, numbers[i]);
  }
 return 0;
}
```

Output:

```
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50
```

Initializing One Dimensional Array in C

In C programming language, we can initialize a one-dimensional array while declaring it or later in the program. We can initialize a one-dimensional array while declaring it by using the following

syntax: **dataType arrayName[arraySize] = {element1, element2, ..., elementN};**

Example of Initializing One Dimensional Array in C

```
#include <stdio.h>
int main()
 {
 int numbers[5] = {10, 20, 30, 40, 50};
 for(int i=0; i<5; i++)
 {
printf("numbers[%d] = %d\n", i, numbers[i]);
}
return 0;
}
```

Output of the code:

```
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50
```

Accessing Elements of One-Dimensional Array in C

In a one-dimensional array, each element is identified by its ***index*** or ***position*** in the array. The index of the first element in the array is ***0***, and the index of the last element is ***arraySize - 1***.

To access an element of a one-dimensional array in C programming language, we use the following

syntax: **arrayName[index]**

arrayName is the name of the array.

index is the index of the element we want to access.

Example of Accessing Elements of One-Dimensional Array in C

```
#include <stdio.h>
 int main()
 {
 int numbers[5] = {10, 20, 30, 40, 50};
printf("The first element of the array is: %d\n", numbers[0]);
printf("The third element of the array is: %d\n", numbers[2]);
return 0;
}
```

Output:

```
The first element of the array is: 10
The third element of the array is: 30
```

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. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

Declaration of two dimensional Array in C

The syntax to declare the 2D array is : **data_type array_name[rows][columns];**

Consider the following example : **int** twodimen[4][3];

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays.

At Compile time: Int a[2][3]={0,0,0,1,1,1}; or {{0,0,0},{1,1,1}};

```c
#include<stdio.h>
int main()
{
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++)
{
 for(j=0;j<3;j++)
{
printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
 }//end of j
}//end of i
return 0;
}
```

Output:

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

Run time:

```c
int a[2][3];
for(i=0;i<2;i++)
{
for(j=0;j<3;j++)
{
scanf(“%d”,&a[i][j]);
}
#include <stdio.h>
void main ()
{
 int arr[3][3],i,j;
 for (i=0;i<3;i++)
{
 for (j=0;j<3;j++)
{
printf("Enter a[%d][%d]: ",i,j);
scanf("%d",&arr[i][j]);
 }
}
printf("\n printing the elements ....\n");
 for(i=0;i<3;i++)
{
printf("\n");
for (j=0;j<3;j++)
{
printf("%d\t",arr[i][j]);
 }
 }
}
```

Output

```
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34
Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78
printing the elements ....
56    10    30
34    21    34
45    56    78
```

Sample programs on matrix addition, subtraction, and matrix multiplication

Program to Add Two Matrices

```
#include <stdio.h>
int main()
{
int r, c, a[r][c], b[r][c], sum[r][c], i, j;
 printf("Enter the number of rows : ");
scanf("%d", &r);
printf("Enter the number of columns : ");
scanf("%d", &c);
printf("\nEnter elements of 1st matrix:\n");
 for (i = 0; i < r; i++)
for (j = 0; j < c; j++)
{
scanf("%d", &a[i][j]);
}
printf("Enter elements of 2nd matrix:\n");
for (i = 0; i < r; i++)
for (j = 0; j < c; j++)
{
scanf("%d", &b[i][j]);
 }
// adding two matrices
printf("\nSum of two matrices: \n");
for (i = 0; i < r; ++i)
for (j = 0; j < c; ++j)
 {
sum[i][j] = a[i][j] + b[i][j];
printf("%d\t",c[i][j]);
}
 Printf("\n");
}
```

```
Enter the number of rows : 2
Enter the number of columns : 3

Enter elements of 1st matrix:
Enter element a11: 2
Enter element a12: 3
Enter element a13: 4
Enter element a21: 5
Enter element a22: 2
Enter element a23: 3
Enter elements of 2nd matrix:
Enter element b11: -4
Enter element b12: 5
Enter element b13: 3
Enter element b21: 5
Enter element b22: 6
Enter element b23: 3
```

```
Sum of two matrices:
-2   8   7

10   8   6
```

Addition of two matrices:

```
#include<stdio.h>
int main ()
int a[2][3],b[2][3],c[2][3],i,j;
printf("Enter first matrix:\n");
for(i=0;i<2;i++)
{
for(j=0;j<3;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Enter second matrix:\n");
for(i=0;i<2;i++)
{
for(j=0;j<3;j++)
{
scanf("%d",&b[i][j]);
}
}
printf("sum of two matrices:\n");
for(i=0;i<2;i++)
{
for(j=0;j<3;j++)
{
c[i][j]=a[i][j]+b[i][j];
printf("%d\t",c[i][j]);
}
printf("\n");
}
return 0;
}
```

Output:
Enter first matrix:
1 2 3 4 5 6
Enter second matrix:
6 7 8 9 1 2
sum of two matrices:
7 9 11
13 6 8

Multiplication of two matrices:

```
#include<stdio.h>
int main ()
int a[3][3],b[3][2],c[3][2],i,j,sum;
printf("Enter the elements of first matrix :\n");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Enter the elements of second matrix :\n");
for(i=0;i<3;i++)
{
for(j=0;j<2;j++)
{
scanf("%d",&b[i][j]);
}
```

```
}
printf(“printing the first matrix:\n”);
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf(“%d”,a[i][j]);
}
printf(“\n”);
}
printf(“printing second matrix :\n”);
for(i=0;i<3;i++)
{
for(j=0;j<2;j++)
{
printf(“%d”,b[i][j]);
}
printf(“\n”);
}
printf(“Multiplication of two matrices:\n”);
for(i=0;i<3;i++)
{
for(j=0;j<2;j++)
{
sum =0;
for(k=0;k<3<k++)
{
sum=sum+a[i][k]*b[k][j];
}
c[i][j]=sum;
printf(“%d\t”,c[i][j]);
}
printf(“\n”);
}
return 0;
}
```

Output:

```
Enter elements of first matix:
 0 5 2 -1 1 0 3 5 7 5
Enter elements of second matix:
1 2 0 1 2 5 4
Printing the  first matrix:

 0  5  2
-1  1  0
 3  7  5

Printing the  first matrix:
1  2
0  1
5  4
Multiplication of two matrices:
10  13
-1  -1
28  33
```

Reordering an array in Ascending order.

```
#include<stdio.h>
int main()
{
int i,j,n,a[10],temp;
printf(“Enter no.of elements:”);
scanf(“%d”,&n);
printf(“Enter the elements:”);
for(i=0;i<n,i++)
{
scanf(“%d”,&a[i]);
```

```
}
for(i=0;i<n;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j] > a[j+1])
{
temp= a[j];
a[j]=a[j+1];
a[j+1] = temp;
}
}
}
printf("the list is :\n");
for(i=0;i<n;i++)
{
printf("%d",a[i]);
}
return 0;
}
```

Output:

```
Enter the number of elements in the list :5
Enter the elements:
23 56 34 98 21
The list is
21 23 34 56 98
```

Strings

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'.Character strings are often used to build meaningful and readable programs.

C String Declaration Syntax

Declaring a string in C is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

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, i.e the number of characters strings will store.

There is an extra terminating character which is the ***Null character ('\0') used to indicate the termination of a string that differs strings from normal character arrays***.

C String Initialization

A string in C can be initialized in different ways. We will explain this with the help of an example. Below are the examples to declare a string with the name str and initialize it with "programming".

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[] = "programming";
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. If we want to store a string of size n then we should always declare a string with a size equal to or greater than n+1.
   char str[50] = "programming";
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] = { 'p','r','o','g','r','a','m','m','i','n','g','\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[] = { 'p','r','o','g','r','a','m','m','i','n','g','\0'};

Reading and displaying of strings from terminal with sample program and Puts() and gets()

puts() vs printf() for printing a string

In C, both **puts()** and **printf()** functions are used for printing a string on the console and are defined in <stdio.h> header file.
puts() Function
The **puts()** function is used to write a string to the console and it automatically adds a new line character '\n' at the end.
Syntax: puts("*str*");
printf() Function
The **printf()** function is also used to print data on the console, but it can also be used to print the formatted data to the console based on a specified format string.
Syntax :printf("*format_string*", *Arguments*);
puts() can be preferred for printing a string because it is generally less costly(implementation of puts() is generally simpler than printf()), and if the string has formatting characters like '**%s**', then printf() would give unexpected results. Also, if str is a user input string, then the use of printf() might cause security issues.

Difference between the printf() and puts()

Following are some differences between printf() and puts() functions in C:

printf()	puts
printf allows us to print formatted strings using format specifiers.	puts do not support formatting.
printf does not add new line characters automatically.	puts automatically adds a new line character.
It returns the number of characters successfully written to the console.	It returns a non-negative value on success and EOF (end-of-file) on failure.
printf can handle multiple strings at one time which helps in concatenating the strings in the output.	puts can print a single string at one time.
printf can print data of different data types.	puts can print only strings.

```
// C program to see how scanf()
// stops reading input after whitespaces

#include <stdio.h>
int main()
{
char str[20];
printf("enter something\n");
scanf("%s", str);
printf("you entered: %s\n", str);
return 0;
}
```

Difference between scanf() and gets() in C

Scanf()

It is used to read the input(character, string, numeric data) from the standard input(keyboard).It is used to read the input until it encounters a whitespace, newline or End Of File(EOF).

```
// C program to see how scanf()
// stops reading input after whitespaces

#include <stdio.h>
int main()
{
char str[20];
printf("enter something\n");
scanf("%s", str);
printf("you entered: %s\n", str);
return 0;
}
```

Input: c programming
Output: c

Input: Computer science
Output: Computer

gets() : It is used to read input from the standard input(keyboard).It is used to read the input until it encounters newline or End Of File(EOF).

```
// C program to show how gets()
// takes whitespace as a string.

#include <stdio.h>
int main()
{
char str[20];
printf("enter something\n");
gets(str);
printf("you entered : %s\n", str);
return 0;
}
```

Input: C programming
Output: C programming
Input: Computer science
Output: Computer science

scanf()	gets()
when scanf() is used to read string input it stops reading when it encounters whitespace, newline or End Of File	when gets() is used to read input it stops reading input when it encounters newline or End Of File. It does not stop reading the input on encountering whitespace as it considers whitespace as a string.
It is used to read input of any datatype Its syntax is -: **scanf(const char *format, ...)** It is fast to take input.	It is used only for string input. Its syntax is -: **char *gets(char *str)** It takes one parameter that is the pointer to an array of chars
Format specifiers is used inside scanf to take input.	Its return value is str on success else NULL.

C String Handling 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 : 1.**dest**: Destination string . 2 **src**: Source string

Return value :The strcat() function returns a pointer to the dest string.

```
// 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;
}
```
dest Before: This is an
dest After: This is an example

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 : 1.str: It represents the string variable whose length we have to find.
Return Value : strlen() function in C returns the length of the string.

```
// 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 "Programming"
char str[] = "Programing";
// 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: Programming
Length: 11

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 : 1. str1: This is the first string to be compared. 2. **str2**: This is the second string to be compared.
Return Value :
1.If str1 is less than str2, the return value is less than 0.
2.If str1 is greater than str2, the return value is greater than 0.
3.If str1 is equal to str2, the return value is 0.

```
// 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
```

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*);

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 : **1. dest**: Pointer to the destination array where the content is to be copied. 2 **.src:** string which will be copied.
Return Value 1. strcpy() function returns a pointer pointing to the output string.

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

int main()
{
// defining strings
char source[] = "Programming";
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: Programming
Destination: Programming
```

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*);

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*);

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*);

What is Character Arithmetic?

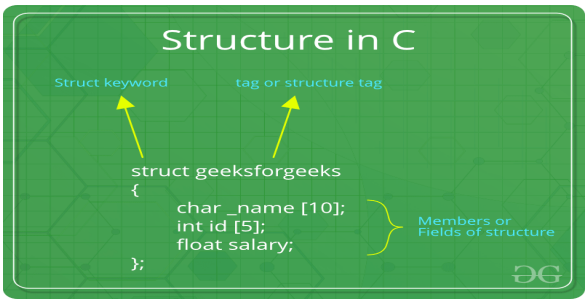
Character arithmetic is used to implement arithmetic operations like addition, subtraction, multiplication, and division on characters in C language. In character arithmetic character converts into an integer value to perform the task. For this ASCII value is used.It is used to perform actions on the strings.

```
// C program to demonstrate character arithmetic.
#include <stdio.h>
int main()
{
char ch1 = 125, ch2 = 10;
ch1 = ch1 + ch2;
printf("%d\n", ch1);
printf("%c\n", ch1 - ch2 - 4);
return 0;
}
```

Output-121
y
So %d specifier causes an integer value to be printed and %c specifier causes a character value to printed. But care has to taken that while using %c specifier the integer value should not exceed 127.

Structures

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The struct keyword is used to define the structure in the C programming language. The items in the structure are called its member and they can be of any valid data type.



To create a structure we may dollow the syntax:

1. The struct keyword.
2. The structure tag_name/
3. List of variables separated by a comma.
4. The template is terminated with a semi colon(;)
5. While the Entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.

C Structure Declaration

In structure declaration, we specify its member variables along with their datatype. We can use the struct keyword to declare the structure in C using the following syntax: Syntax struct structure_name { data_type member_name1; data_type member_name1; .... }; The above syntax is also called a structure template or structure prototype and no memory is allocated to the structure in the declaration.

C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

Structure Variable Declaration with Structure Template

```
struct structure_name
{
data_type member_name1;
```

```
 data_type member_name1; .... ....
 }variable1, varaible2, ...;
```
2.Structure Variable Declaration after Structure Template
```
// structure declared beforehand
struct structure_name variable1, variable2, .......;
```

Access Structure Members

We can access structure members by using the **(.) dot operator.**

Syntax : structure_name.member1;
strcuture_name.member2;

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

```
#include<stdio.h>
struct student
{
int roll_no;
char name[30];
float marks;
}
int  main ()
{
struct student s1={1 ,"Arya", 45.5};
struct student s2={5,"Surya",50.5};
printf("Info of s1\n:");
printf("%d %s %f",s1.roll_no,s1.name,s1.marks);
printf("Info of s2\n:");
printf("%d %s %f",s2.roll_no,s2.name ,s2.marks);
}
```

Out put:

```
Info of s1:
1 arya 45.5
Info of s2:
5 surya 50.5
```

Initialize Structure Members

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

```
struct Point
{
 int x = 0;  // COMPILER ERROR:  cannot initialize members here
 int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator

struct *structure_name* str;
str.member1 = value1;
str.member2 = value2;
str.member3 = value3;
.
.
.

2. Initialization using Initializer List

struct *structure_name* str = { value1, value2, value3 };

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

```
// C program to illustrate the use of structures
#include <stdio.h>
// declaring structure with name str1
struct str1
 {
  int i;
```

```
  char c;
  float f;
  char s[30];
 };
// declaring structure with name str2
struct str2
 {
 int ii;
char cc;
float ff;
}
 var; // variable declaration with structure template
// Driver code
int main()
{
// variable declaration after structure template
// initialization with initializer list and designated
// initializer list
struct str1 var1 = { 1, 'A', 1.00, "GeeksforGeeks" },
var2;
struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };
// copying structure using assignment operator
var2 = var1;
printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",var1.i, var1.c, var1.f, var1.s);
printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",var2.i, var2.c, var2.f, var2.s);
printf("Struct 3:\n\ti = %d, c = %c, f = %f\n", var3.ii, var3.cc, var3.ff);
return 0;
}
```

Output

```
Struct 1:
 i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
 i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3:
i = 5, c = a, f = 5.000000
```

Structure assignment

struct *structure_name* str;
str.member1 = value1;
str.member2 = value2;
str.member3 = value3;
.
.
.

Size of a structure

We use structures and variables to create variables of large size
The actual size of these variables may change from one machine to another. By using Unary operator “sizeof()” to tell the size of the structure or variable.
Sizeof(struct x)

```
#include<stdio.h>
Struct  student
{
Int  pin_no;
Char name;
} s;
Int main()
{
Printf(“size of the structure is %d”,sizeof(s));
}
```

Nested Structures

C language allows us to insert one structure into another as a member. This process is called nesting and such structures are called nested structures. There are two ways in which we can nest one structure into another:

1. **Separate Structure Nesting**
   In this method two structures are created. but the dependent structure should be used inside the main structure as a member.
   #include<stdio.h

```
#include<string.h>
struct employee
{
int employee_id;
char name[20];
float salary;
}
struct organization
{
char org_name[50];
int org_num;
struct employee emp;
};
int main ()
{
struct organization org;
org.employee_id = 101;
strcpy(org.emp.name,"A rjun");
org.emp.salary =19000.0;
strcpy(org.org_name,"abcd");
org.org_num=295;
printf("organization name :%s",org.org_name);
printf("organization number :%d",org.org_num);
printf("Employee id; %d",org.emp.employeee_id);
printf("Employee name: %s", org.emp.name);
printf("Employee salary:%f",org.emp.salary);
}
Output:
organization name : abcd
organization number:295
Employee id: 101
Employee name: Arjun
Employee salary:19000.0
```

2.Embedded Structure Nesting:

Whenever an embedded nested structure is created, the variable declaration is compulasary at the end of the inner structure,which acts as the member of the outer structure.

```
#include<stdio.h
#include<string.h>
struct organization
{
char org_name[50];
int org_num;
struct employee
{
int employee_id;
char name[20];
float salary;
} emp;
};
int main ()
{
struct organization org;
org.employee_id = 101;
strcpy(org.emp.name,"A rjun");
org.emp.salary =19000.0;
strcpy(org.org_name,"abcd");
org.org_num=295;
printf("organization name :%s",org.org_name);
printf("organization number :%d",org.org_num);
printf("Employee id; %d",org.emp.employeee_id);
printf("Employee name: %s", org.emp.name);
printf("Employee salary:%f",org.emp.salary);
}
Output:
organization name : abcd
organization number:295
```

Employee id: 101
Employee name: Arjun
Employee salary:19000.0

Structure containing arrays and Array of structures :

Array within a Structure or structure containing array
A structure may contain elements of different data types – int, char, float, double, etc. It may also contain an array as its member. Such an array is called an array within a structure. An array within a structure is a member of the structure and can be accessed just as we access other elements of the structure.

```
// C program to demonstrate the
// use of an array within a structure
#include <stdio.h>
// Declaration of the structure candidate
struct candidate
{
int roll_no;
char grade;
// Array within the structure
float marks[4];
};
// Function to displays the content of
// the structure variables
void display(struct candidate a1)
{
printf("Roll number : %d\n", a1.roll_no);
printf("Grade : %c\n", a1.grade);
printf("Marks secured:\n");
int i;
int len = sizeof(a1.marks) / sizeof(float);
// Accessing the contents of the
// array within the structure
for (i = 0; i < len; i++)
{
printf("Subject %d : %.2f\n", i + 1, a1.marks[i]);
}
}
// Driver Code
int main()
{
// Initialize a structure
struct candidate A = { 1, 'A', { 98.5, 77, 89, 78.5 } };
// Function to display structure
display(A);
return 0;
}
```

Output
Roll number : 1
Grade : A
Marks secured:
Subject 1 : 98.50
Subject 2 : 77.00
Subject 3 : 89.00
Subject 4 : 78.50

Array of Structures

An array is a collection of data items of the same type. Each element of the array can be int, char, float, double, or even a structure. We have seen that a structure allows elements of different data types to be grouped together under a single name.

```
// C program to demonstrate the
// usage of an array of structures
#include <stdio.h>
// Declaring a structure class
struct class
 {
int roll_no;
char grade;
float marks;
```

```
};
// Function to displays the contents
// of the array of structures
void display(struct class class_record[3])
{
int i, len = 3;
// Display the contents of the array
// of structures here, each element
// of the array is a structure of class
for (i = 0; i < len; i++)
{
printf("Roll number : %d\n",class_record[i].roll_no);
printf("Grade : %c\n", class_record[i].grade);
printf("Average marks : %.2f\n",class_record[i].marks);
printf("\n");
}
}
// Driver Code
int main()
{
// Initialize of an array of structures
struct class class_record[3] = { { 1, 'A', 89.5f },{ 2, 'C', 67.5f },{ 3, 'B', 70.5f } };
// Function Call to display
// the class_record
display(class_record);
return 0;
}
```

Output

```
Roll number : 1
Grade : A
Average marks : 89.50
Roll number : 2
Grade : C
Average marks : 67.50
Roll number : 3
Grade : B
Average marks : 70.50
```

Difference between Array of Structures and Array within Structures

Below is the tabular difference between the Array within a Structure and Array of Structures:

Parameter	Array within a Structure	Array of Structures
Basic idea	A structure contains an array as its member variable.	An array in which each element is of type structure.
Syntax	struct class { int ar[10]; } a1, a2, a3;	struct class { int a, b, c; } students[10];
Access	Can be accessed using the dot operator just as we access other elements of the structure.	Can be accessed by indexing just as we access an array.
Access elements syntax	structure.array[index]	array[index].member
Memory Structure	Array within the structure will be stored in sequential memory and structure padding is not dependent on the size of the array.	There will be some empty space between structure elements due to structure padding.

C Unions

The Union is a user-defined data type in C language that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given instance.

Syntax of Union in C

union tag_name
{
Type_1 member_1;
Type_2 member_2;
Type_n member_n;
};

The syntax of the union in C can be divided into three steps which are as follows:

C Union Declaration

In this part, we only declare the template of the union, i.e., we only declare the members' names and data types along with the name of the union. No memory is allocated to the union in the declaration.

union *union_name*
{
datatype member1;
datatype member2;
...
};

Different Ways to Define a Union Variable

We need to define a variable of the union type to start using union members. There are two methods using which we can define a union variable.

1.With Union Declaration
2. After Union Declaration

1. Defining Union Variable with Declaration

union **union_name**
{
datatype member1;
datatype member2;
...
} var1, var2, ...;

2. Defining Union Variable after Declaration

union ***union_name var1, var2, var3...;***
where *union_name* is the name of an already declared union.

Access Union Members

We can access the members of a union by using the (.) dot operator just like structures.
var1.member1;
where *var1* is the **union variable** and *member1* is the **member of the union**.
The above method of accessing the members of the union also works for the nested unions.
var1.member1.memberA;
Here,

- *var1* is a union variable.
- *member1* is a member of the union.
- *memberA* is a member of member1.

Initialization of Union in C

- The initialization of a union is the initialization of its members by simply assigning the value to it.
- *var1.member1 = some_value*
- One important thing to note here is that **only one member can contain some value at a given instance of time.**

// C Program to demonstrate how to use union
#include <stdio.h>
// union template or declaration

```
union un
{
int member1;
char member2;
float member3;
};
// driver code
int main()
{
// defining a union variable
union un var1;
// initializing the union member
var1.member1 = 15;
printf("The value stored in member1 = %d",var1.member1);
return 0;
}
```

Output

The value stored in member1 = 15

Need of union

It occupies less memory compared to the structure. When you use union, only the last variable can be directly accessed. Union is used when you have to use the same memory location for two or more data members.

Size of Union

The size of the union will always be equal to the size of the largest member of the array. All the less-sized elements can store the data in the same space without any overflow.

Differences between Structure and Union

Differences between Structure and Union are as shown below in tabular format as shown below as follows:

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.