Strings

In **C Language, string** is a character sequence terminated with a null character ('\0', called NUL in ASCII). It is usually stored as one-dimensional character array.

Strings are arrays of chars. String literals are words surrounded by double quotation marks.

```
"This is a static string"
```

To declare a string of 49 letters, you would want to say:

```
char string[50];
```

This would declare a string with a length of 50 characters. Do not forget that arrays begin at zero, not 1 for the index number. In addition, a string ends with a null character, literally a '\0' character. However, just remember that there will be an extra character on the end on a string. It is like a period at the end of a sentence, it is not counted as a letter, but it still takes up a space. Technically, in a fifty char array you could only hold 49 letters and one null character at the end to terminate the string.

The format specifier used in case of string is *%s*

Example: Program to enter and display a string

```
#include<stdio.h>
void main()
{
  char name[25];

  printf("\nEnter your name");

  scanf("%s",&name);

  printf("Name you enetered is : %s", name);
}
```

Functions used to handle strings in C Language

- **strlen()**
- **strcpy()**
- **strncpy()**
- **strcat()**
- **strncat()**
- **strcmp()**
- **strncmp()**

---

strlen()

Syntax: len = strlen(ptr);
where len is an integer and
ptr is a pointer to char

strlen() returns the length of a string, excluding the null. The following code will result in *len* having the value 13.

```
int len;
char str[15];

strcpy(str, "Hello, world!");
len = strlen(str);
```

---

strcpy()

Syntax: strcpy(ptr1, ptr2);
where ptr1 and ptr2 are pointers to char

strcpy() is used to copy a null-terminated string into a variable. Given the following declarations, several things are possible.

```
char S[25];
char D[25];
```

- Putting text into a string:
- strcpy(S, "This is String 1.");

- Copying a whole string from S to D:
- strcpy(D, S);

- Copying the tail end of string S to D:
- strcpy(D, &S[8]);

---

strncpy()

Syntax: strncpy(ptr1, ptr2, n);

where n is an integer and
ptr1 and ptr2 are pointers to char

strncpy() is used to copy a portion of a possibly null-terminated string into a variable. Care must be taken because the '\0' is put at the end of destination string *only* if it is within the part of the string being copied. Given the following declarations, several things are possible.

```
char S[25];
char D[25];
```

Assume that the following statement has been executed before each of the remaining code fragments.

- Putting text into the source string:
- strcpy(S, "This is String 1.");

- Copying four characters from the beginning of S to D and placing a null at the end:
- strncpy(D, S, 4);
- D[4] = '\0';

- Copying two characters from the middle of string S to D:
- strncpy(D, &S[5], 2);
- D[2] = '\0';

- Copying the tail end of string S to D:
- strncpy(D, &S[8], 15);
  which produces the same result as strcpy(D, &S[8]);

---

strcat()

Syntax: strcat(ptr1, ptr2);
where ptr1 and ptr2 are pointers to char

strcat() is used to concatenate a null-terminated string to end of another string variable. This is equivalent to pasting one string onto the end of another, overwriting the null terminator. There is only one common use for strcat().

```
char S[25] = "world!";
char D[25] = "Hello, ";
```

- Concatenating the whole string S onto D:
- strcat(D, S);

---

strcmp()

Syntax: diff = strcmp(ptr1, ptr2);
where diff is an integer and
ptr1 and ptr2 are pointers to char

strcmp() is used to compare two strings. The strings are compared character by character starting at the characters pointed at by the two pointers. If the strings are identical, the integer value zero (0) is returned. As soon as a difference is found, the comparison is halted and if the ASCII value at the point of difference in the first string is less than that in the second (e.g. 'a' 0x61 vs. 'e' 0x65) a negative value is returned; otherwise, a positive value is returned. Examine the following examples.

```
char s1[25] = "pat";
char s2[25] = "pet";
```

diff will have a *negative* value after the following statement is executed.

```
diff = strcmp(s1, s2);
```

diff will have a *positive* value after the following statement is executed.

```
diff = strcmp(s2, s1);
```

diff will have a value of *zero* (0) after the execution of the following statement, which compares *s1* with itself.

```
diff = strcmp(s1, s1);
```

---

strncmp()

Syntax: diff = strncmp(ptr1, ptr2, n);
where diff and n are integers
ptr1 and ptr2 are pointers to char

strncmp() is used to compare the first n characters of two strings. The strings are compared character by character starting at the characters pointed at by the two pointers.

If the first n strings are identical, the integer value zero (0) is returned. As soon as a difference is found, the comparison is halted and if the ASCII value at the point of difference in the first string is less than that in the second (e.g. 'a' 0x61 vs. 'e' 0x65) a negative value is returned; otherwise, a positive value is returned. Examine the following examples.

```
char s1[25] = "pat";
char s2[25] = "pet";
```

diff will have a *negative* value after the following statement is executed.

```
diff = strncmp(s1, s2, 2);
```

diff will have a *positive* value after the following statement is executed.

```
diff = strncmp(s2, s1, 3);
```

diff will have a value of *zero* (0) after the following statement.

```
diff = strncmp(s1, s2, 1);
```

Example: Program to reverse the string without using strrev() function

```
#include<stdio.h>
#include<string.h>
void main()
{

char str[50] revstr[50];
int i=0 j=0;

printf( Enter the string to be reversed : );
scanf( "%s",&str);

for(i=strlen(str)-1;i>=0;i--)
{
revstr[j]=str[i];
j++;
}

revstr[j]='\0';
printf( "Input String : %s", str);
printf( "\nOutput String : %s", revstr);
getch();
}
```

Example: Program to reverse the string using strrev() function

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

int main()
{
   char arr[100];

   printf("Enter a string to reverse ");
   gets(arr);

   strrev(arr);

   printf("Reverse of entered string is %s\n",arr);

   return 0;
}
```

Example: Program to check whether a string is palindrome or not

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

#define size 26

void main()
{
char strsrc[size];
char strtmp[size];

printf("\n Enter String:= ");
gets(strsrc);

strcpy(strtmp,strsrc);
strrev(strtmp);

if(strcmp(strsrc,strtmp)==0)
  printf("\n Entered string \"%s\" is palindrome",strsrc);
else
  printf("\n Entered string \"%s\" is not palindrome",strsrc);

}
```

Structures

Introducing to C structure

In some programming contexts, you need to access multiple data types under a single name for easier data manipulation; for example you want to refer to address with multiple data like house number, street, zip code, country. C supports structure which allows you to wrap

one or more variables with different data types. A structure can contain any valid data types like int, char, float even arrays or even other structures. Each variable in structure is called a structure member.

Defining structure

To define a structure, you use ***struct*** keyword. Here is the common syntax of structure

definition:
struct struct_name{ structure_member };

The name of structure follows the rule of variable name. Here is an example of defining ***address*** structure:

```
struct address{
        unsigned int house number;
        char street_name[50];
        int zip_code;
        char country[50];
};
```

The ***address*** structure contains house number as an positive integer, street name as a string, zip code as an integer and country as a string.

Declaring structure

The above example only defines an ***address*** structure without creating any structure instance. To create or declare a structure instance, you can do it in two ways:
The first way is to declare a structure followed by structure definition like this:

```
struct struct_name{
    Data_type variable;  // structure member
                .
                .
} instance 1, instance 2…..,instance n;
```

In the second way, you can declare the structure instance at a different location in your source code after structure definition. Here is structure declaration syntax :

struct struct_name instance_1,instance_2 instance_n;

Accessing structure member

To access structure members we can use dot operator (.) between structure name and structure member name as follows:

structure_name*.structure_member*

For example to access street name of structure *address* we do as follows:

struct address billing_addr;
billing_addr.country=”India”;

Initializing structures

C programming language treats a structure as a custom data type therefore you can initialize a structure like a variable. Here is an example of initialize *product* structure:

```
struct product{
        char name[50];
        double price;
} book = { “ C programming language”,40.5};
```

In above example, we define product structure, then we declare and initialize book structure with its name and price.

Structure and pointer

A structure can contain pointers as structure members and we can create a pointer to a structure as follows:

```
struct person{
        char name[50];
        unsigned int age;
};

struct person p, *ptr;
```

Shorthand structure with *typedef* keyword

To make your source code more concise, you can use *typedef* keyword to create a synonym for a structure. This is an example of using *typedef* keyword to define address structure so when you want to create an instance of it you can omit the keyword *struct*

```
typedef struct{
        unsigned int house number;
        char street_name[50];
        int zip_code;
        char country[50];
}address;

address permanent;
```

Copy a structure into another structure

One of major advantage of structure is you can copy it with = operator. The syntax as follows

```
typedef struct{
        unsigned int house number;
        char street_name[50];
        int zip_code;
        char country[50];
}address;

address a1,a2;

a1=a2;
```

Noted: Some old C compilers may not supports structure assignment so you have to assign each member variables one by one.

Structure and sizeof function

sizeof is used to get the size of any data types even with any structures. Let's take a look at simple program:

```
#include <stdio.h>

typedef struct __address{
    int house_number;// 4 bytes
    char street[50]; // 50 bytes
```

```
    int zip_code; // 4 bytes
    char country[20];// 20 bytes

} address;//78 bytes in total

void main()
{
    // it returns 80 bytes
    printf("size of address is %d bytes\n",sizeof(address));
}
```

Array of Structure

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example:

```
structure information
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}student[100];
```

An array of structures can be assigned initial values just as any other array can. Remember that each element is a structure that must be assigned corresponding initial values as illustrated below.

```
#include <stdio.h>

void main()
{

struct info{
```

```
int id_no;
char name[20];
char address[20];
int age;
}

struct info std[100];
int I,n;

printf("Enter the number of students");
scanf("%d",&n);

printf(" Enter Id_no, name, address, age ");
for(I=0;I < n;I++)
scanf(%d%s%s%d",&std[I].id_no, std[I].name, std[I].address, &std[I].age);

printf("\n Student information");
for (I=0;I< n;I++)
printf("%d%s%s%s%dn", ",std[I].id_no, std[I].name, std[I].address, std[I].age);

getch();
}
```

Structures within structures (Nested Structures)

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
struct date{
int day;
int month;
int year;
};
```

```
struct student{
int id_no;
char name[20];
char address[20];
int age;
structure date def;
structure date doa;
}oldstudent, newstudent;
```

Union

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computers memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to conserve memory. They are useful for application involving multiple members. Where values need not be assigned to all the members at any one time. Like structures union can be declared using the keyword union as follows:

```
union item{
        int m;
        float p;
        char c;
}code;
```

this declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is because if only one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

```
code.m
code.p
code.c
```

are all valid member variables. During accessing we should make sure that we are accessing the member whose value is currently stored.
For example a statement such as:

```
code.m=456;
code.p=456.78;
printf("%d", code.m);
```

Would produce erroneous result.

In effect a union creates a storage location that can be used by one of its members at a time. When a different number is assigned a new value the new value supercedes the previous members value. Unions may be used in all places where a structure is allowed. The notation for accessing a union member that is nested inside a structure remains the same as for the nested structure.

Difference between Structure & Union

UNIONS	STRUCTURES
1. It can hold different types (variables) in a single location.	1.It can hold different types (variables) indifferent locations
2. It may contain more than one type (variable) but only one is stored at a time.	2. It may contain more than one type (variable) all are stored in memory at same time.
3. union type variable takes the largest memory occupied by its member	3. It requires memory of the size of all its members.

4. Union declaration syntax
union person{
char name[50];
int age;
};
	4. Structure declaration syntax
	struct person{
	char name[50];
	int age;
	};