const Pointer in C

Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax of Constant Pointer

1. <type of pointer> ***const*** <name of pointer>;

Declaration of a constant pointer is given below:

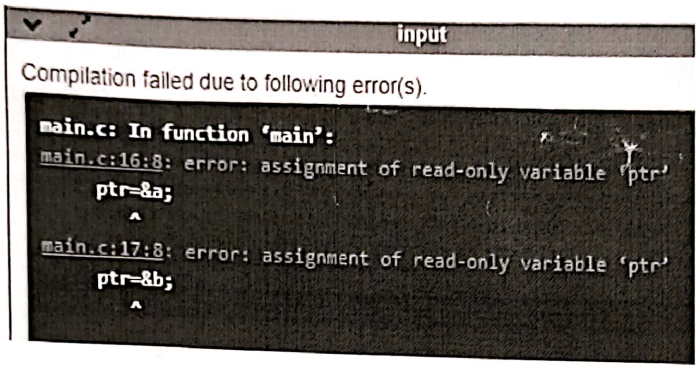
```
int *const ptr;
```

```
#include <stdio.h>
int main()
{
    int a=1;
    int b=2;
    int *const ptr;
    ptr=&a;
    ptr=&b;
    printf("Value of ptr is :%d",*ptr);
    return 0;
}
```

In the above code:

- We declare two variables, i.e., a and b with values 1 and 2, respectively.
- We declare a constant pointer.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of the variable pointed by the 'ptr'.

Output



In the above output, we can observe that the above code produces the error "assignment of read-only variable 'ptr'". It means that the value of the variable 'ptr' which 'ptr' is holding cannot be changed. In the above code, we are changing the value of 'ptr' from &a to &b, which is not possible with constant pointers. Therefore, we can say that the constant pointer, which points to some variable, cannot point to another variable.

Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

Syntax of Pointer to Constant

const <type of pointer>* <name of pointer>

Declaration of a pointer to constant is given below:

const int* ptr;

Let's understand through an example.

- **First, we write the code where we are changing the value of a pointer**

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=100;
5.     int b=200;
6.     const int* ptr;
7.     ptr=&a;
8.     ptr=&b;
9.     printf("Value of ptr is :%u",ptr);
10.    return 0;
11.}
```

In the above code:

- We declare two variables, i.e., a and b with the values 100 and 200 respectively.
- We declare a pointer to constant.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of 'ptr'.

Output

```
Value of ptr is :247760772
```

The above code runs successfully, and it shows the value of 'ptr' in the output.

Pointers in C Programming with examples

A **pointer** is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of an integer variable.

Understand what we mean by the address of a variable?

A simple example to understand how to access the address of a variable without pointers?

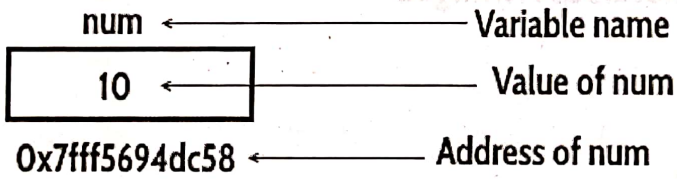
In this program, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let's say the address assigned to variable num is 0x7fff5694dc58, which means whatever value we would be assigning to num should be stored at the location: 0x7fff5694dc58. See the diagram below.

```
#include <stdio.h>
int main()
{
   int num = 10;
   printf("Value of variable num is: %d", num);
   /* To print the address of a variable we use %p
    * format specifier and ampersand (&) sign just
    * before the variable name like &num.
    */
   printf("\nAddress of variable num is: %p", &num);
   return 0;
}
```

Output:

```
Value of variable num is: 10
Address of variable num is: 0x7fff5694dc58
```



A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

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

```
//Variable declaration
int num = 10;

//Pointer declaration
int *p;

//Assigning address of num to the pointer p
p = #

printf("Address of variable num is: %p", p);
return 0;
}
```

Output:

```
Address of variable num is: 0x7fff5694dc58
```

C Pointers – Operators that are used with Pointers

Lets discuss the operators & and * that are used with Pointers in C.

"Address of"(&) Operator

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The **& operator** is also known as "**Address of**" Operator.

```
printf("Address of var is: %p", &num);
```

Point to note: %p is a format specifier which is used for displaying the address in hex format.

Now that you know how to get the address of a variable but **how to store that address in some other variable?** That's where pointers comes into picture. As mentioned in the beginning of this guide, pointers in C programming are used for holding the address of another variables.

Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.

"Value at Address"(*) Operator

The * Operator is also known as **Value at address** operator.

How to declare a pointer?

```
int *p1  /*Pointer to an integer variable*/
double *p2  /*Pointer to a variable of data type double*/
char *p3   /*Pointer to a character variable*/
float *p4   /*pointer to a float variable*/
```

The above are the few examples of pointer declarations. **If you need a pointer to store the address of integer variable then the data type of the pointer should be int**. Same case is with the other data types.

By using * operator we can access the value of a variable through a pointer.
For example:

```
double a = 10;
double *p;
p = &a;
```

*p would give us the value of the variable a. The following statement would display 10 as output.

```
printf("%d", *p);
```

Similarly if we assign a value to *pointer like this:

```
*p = 200;
```

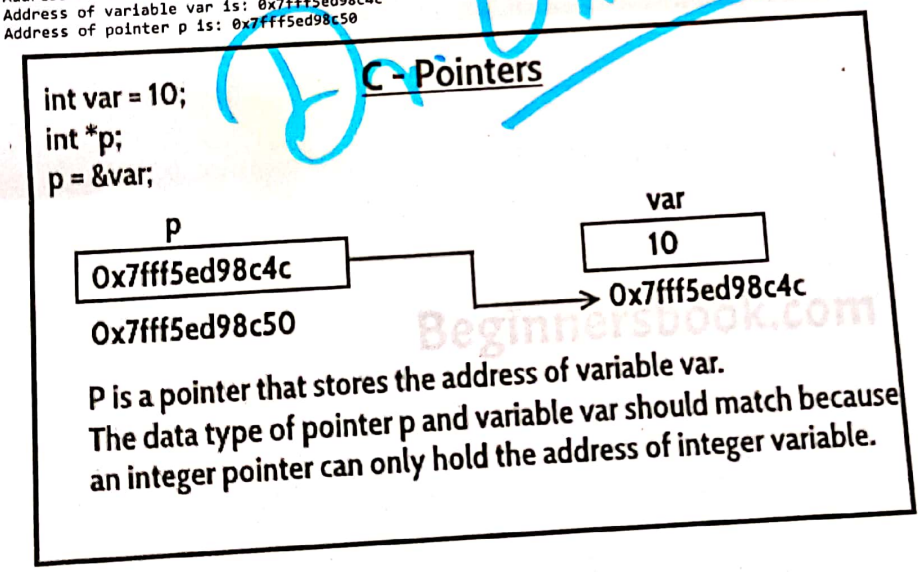
It would change the value of variable a. The statement above will change the value of a from 10 to 200.

Example of Pointer demonstrating the use of & and *

```c
#include <stdio.h>
int main()
{
   /* Pointer of integer type, this can hold the
    * address of a integer type variable.
    */
   int *p;

   int var = 10;

   /* Assigning the address of variable var to the pointer
    * p. The p can hold the address of var because var is
    * an integer type variable.
    */
   p= &var;

   printf("Value of variable var is: %d", var);
   printf("\nValue of variable var is: %d", *p);
   printf("\nAddress of variable var is: %p", &var);
   printf("\nAddress of variable var is: %p", p);
   printf("\nAddress of pointer p is: %p", &p);
   return 0;
}
```

Output:

```
Value of variable var is: 10
Value of variable var is: 10
Address of variable var is: 0x7fff5ed98c4c
Address of variable var is: 0x7fff5ed98c4c
Address of pointer p is: 0x7fff5ed98c50
```



Lets take few more examples to understand it better –
Lets say we have a char variable ch and a pointer ptr that holds the address of ch.

```c
char ch='a';
char *ptr;
```

Read the value of ch

```c
printf("Value of ch: %c", ch);
or
printf("Value of ch: %c", *ptr);
```

Change the value of ch

```c
ch = 'b';
or
*ptr = 'b';
```

The above code would replace the value 'a' with 'b'.

Can you guess the output of following C program?

```
#include <stdio.h>
int main()
{
    int var =10;
    int *p;
    p= &var;

    printf ( "Address of var is: %p", &var);
    printf ( "\nAddress of var is: %p", p);

    printf ( "\nValue of var is: %d", var);
    printf ( "\nValue of var is: %d", *p);
    printf ( "\nValue of var is: %d", *( &var));

    /* Note I have used %p for p's value as it represents an address*/
    printf( "\nValue of pointer p is: %p", p);
    printf ( "\nAddress of pointer p is: %p", &p);

    return 0;
}
```

Output:

```
Address of var is: 0x7fff5d027c58
Address of var is: 0x7fff5d027c58
Value of var is: 10
Value of var is: 10
Value of var is: 10
Value of pointer p is: 0x7fff5d027c58
Address of pointer p is: 0x7fff5d027c50
```

More Topics on Pointers

1) **Pointer to Pointer** – A pointer can point to another pointer (which means it can store the address of another pointer), such pointers are known as double pointer OR pointer to pointer.

2) **Passing pointers to function** – Pointers can also be passed as an argument to a function, using this feature a function can be called by reference as well as an array can be passed to a function while calling.

3) **Function pointers** – A function pointer is just like another pointer, it is used for storing the address of a function. Function pointer can also be used for calling a function in C program.

Void Pointer in C

A pointer in a program that isn't associated with a data type is known as a void pointer in C. The void pointer points to the data location

The void pointer in C is a pointer that is not associated with any data types. It points to some data location in the storage. This means that it points to the address of variables. It is also called the general purpose pointer. In C, malloc() and calloc() functions return void * or generic pointers.

What is a Void Pointer in C?

The void pointers refer to the pointers that have no data type associated with them. As the name suggests, the void pointer indicates that the pointer is basically empty- and thus, it is capable of holding any data type address in the program. Then, these void pointers that have addresses, can be further typecast into other data types very easily.

The allocation of memory also becomes very easy with such void pointers. It is because they make the functions flexible enough to be allocated with bytes and memories appropriately. Let us look at the syntax for the void pointer in C.

Syntax

```
void *name_of_pointer;
```

Here, the *void* keyword acts as the pointer type, and it is followed by the pointer name- to which the pointer type points and allocates the address location in the code. The declaration of a pointer happens with the name and type of pointer that supports any given data type. Let us take a look at an example to understand this better.

```
void *ptr
```

Here, the pointer is expecting a void type- not int, float, etc. It is pointed by the *ptr* pointer here that includes the * symbol. It denotes that this pointer has been declared, and it will be used for dereferencing in the future.

Why use a Void Pointer in C?

As we all know, the address that is assigned to any pointer must have the same data type as we have specified in the declaration of the pointer. For instance, when we are declaring the float pointer, this float pointer won't be able to point to an int variable or any other variable. In simpler words, it will only be able to point to the float type variable. We thus use a pointer to void to overcome this very problem.

The pointer to void can be used in generic functions in C because it is capable of pointing to any data type. One can assign the void pointer with any data type's address, and then assign the void pointer to any pointer without even performing some sort of explicit typecasting. So, it reduces complications in a code.

Example Code

Let us take a look at an example of how we use the above void pointers in a code.

```
#include<stdlib.h>
int main() {
int v = 7;
float w = 7.6;
void *u;
u = &v;
printf("The Integer variable in the program is equal to = %d", *( (int*) u) );
u = &w;
printf("\nThe Float variable in the program is equal to = %f", *( (float*) u) );
return 0;
}
```

The output obtained out of the program would be:

The Integer variable in the program is equal to = 7

The Float variable in the program is equal to = 7.600000

Now, we will take a look at a few more examples to understand how we can use the void pointer in a C program.

Example #1

```
#include<stdio.h>
int main()
{
int b = 52;
void *p = &b;
printf("%d", *(int *)p);
return 0;
}
```

The output obtained out of this program would be:

52

Example #2

```
#include<stdio.h>
int main()
```

```
{
int a[3] = {5,9,7};
void *p = &a;
p = p + sizeof(int);
printf("%d", *(int *)p);
return 0;
}
```

The output obtained out of this program would be:

9

Dr. Uma

NULL pointer in C

A null pointer is a pointer which points nothing.

Some uses of the null pointer are:

a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.

b) To pass a null pointer to a function argument when we don't want to pass any valid memory address.

c) To check for null pointer before accessing any pointer variable. So that, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.

Algorithm

```
Begin.
  Declare a pointer p of the integer datatype.
    Initialize *p= NULL.
  Print "The value of pointer is".
    Print the value of the pointer p.
End.
```

Dr. Uma

Example

```
#include <stdio.h>
int main() {
  int *p= NULL;//initialize the pointer as null.
  printf("The value of pointer is %u",p);
  return 0;
}
```

Output

The value of pointer is 0.

C dereference pointer

As we already know that **"what is a pointer"**, a pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (*). When indirection operator (*) is used with the pointer variable, then it is known as **dereferencing a pointer.** When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Why we use dereferencing pointer?

Dereference a pointer is used because of the following reasons:

- It can be used to access or manipulate the data stored at the memory location, which is pointed by the pointer.
- Any operation applied to the dereferenced pointer will directly affect the value of the variable that it points to.

Let's observe the following steps to dereference a pointer.

- First, we declare the integer variable to which the pointer points.

Dr. Uma

1. **int** x =9;
   - Now, we declare the integer pointer variable.

1. **int** *ptr;
   - After the declaration of an integer pointer variable, we store the address of 'x' variable to the pointer variable 'ptr'.

1. ptr=&x;
   - We can change the value of 'x' variable by dereferencing a pointer 'ptr' as given below:

1. *ptr =8;

The above line changes the value of 'x' variable from 9 to 8 because 'ptr' points to the 'x' location and dereferencing of 'ptr', i.e., *ptr=8 will update the value of x.

Let's combine all the above steps:

```
#include <stdio.h>
1. int main()
2. {
3.    int x=9;
4.    int *ptr;
5.    ptr=&x;
6.    *ptr=8;
7.    printf("value of x is : %d", x);
8.    return 0;}
```

Output

```
value of x is : 8

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's consider another example.

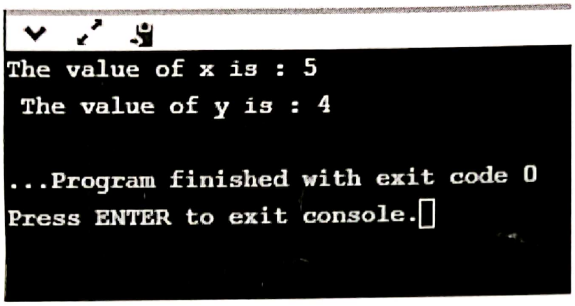
```
1.  #include <stdio.h>
2.  int main()
3.  {
4.     int x=4;
5.     int y;
6.     int *ptr;
7.     ptr=&x;
8.     y=*ptr;
9.     *ptr=5;
10.    printf("The value of x is : %d",x);
11.    printf("\n The value of y is : %d",y);
12.    return 0;
13. }
```

In the above code:

- We declare two variables 'x' and 'y' where 'x' is holding a '4' value.
- We declare a pointer variable 'ptr'.
- After the declaration of a pointer variable, we assign the address of the 'x' variable to the pointer 'ptr'.
- As we know that the 'ptr' contains the address of 'x' variable, so '*ptr' is the same as 'x'.
- We assign the value of 'x' to 'y' with the help of 'ptr' variable, i.e., y=***ptr** instead of using the 'x' variable.

Note: According to us, if we change the value of 'x', then the value of 'y' will also get changed as the pointer 'ptr' holds the address of the 'x' variable. But this does not happen, as 'y' is storing the local copy of value '5'.

Output



Let's consider another scenario.

```
1. #include <stdio.h>
2. int main()
3. {
4.    int a=90;
5.    int *ptr1,*ptr2;
6.    ptr1=&a;
7.    ptr2=&a;
8.    *ptr1=7;
9.    *ptr2=6;
10.   printf("The value of a is : %d",a);
11.   return 0;
12. }
```

In the above code:

- First, we declare an 'a' variable.

- Then we declare two pointers, i.e., ptr1 and ptr2.
- Both the pointers contain the address of 'a' variable.
- We assign the '7' value to the *ptr1 and '6' to the *ptr2. The final value of 'a' would be '6'.

Note: If we have more than one pointer pointing to the same location, then the change made by one pointer will be the same as another pointer.

Output

```
The value of a is : 6
```

C - Array of pointers

Which uses an array of 3 integers –

```
#include <stdio.h>

const int MAX = 3;

int main () {

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

  for (i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, var[i] );
  }

  return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer –

```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows –

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];

  for ( i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; /* assign the address of integer. */
  }
```

```
  for ( i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }

  return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows –

```
#include <stdio.h>

const int MAX = 4;

int main () {

  char *names[] = {
    "Zara Ali",
    "Hina Ali",
    "Nuha Ali",
    "Sara Ali"
  };

  int i = 0;

  for ( i = 0; i < MAX; i++) {
    printf("Value of names[%d] = %s\n", i, names[i] );
  }

  return 0;
}
```

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

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

Example 1: Pointers and Arrays

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

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

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {
  // Equivalent to scanf("%d", &x[i]);
      scanf("%d", x+i);

  // Equivalent to sum += x[i]
      sum += *(x+i);
  }

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

  return 0;
}
```

When you run the program, the output will be:

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```

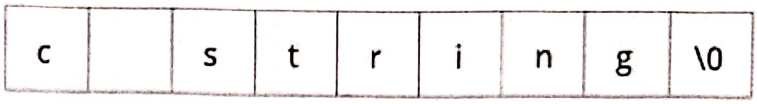
UNIT-3

STRING IN C

In C programming, a string is a sequence of characters terminated with a null character \0. For example:

```
char c[] = "c string";
```

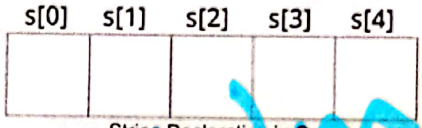
When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.



Memory Diagram

How to declare a string?

```
char s[5];
```



String Declaration in C

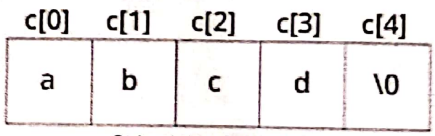
We have declared a string of 5 characters.

Dr. Uma

How to initialize strings?

You can initialize strings in a number of ways.

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



String Initialization in C

Let's take another example:

```
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is `'\0'`) to a `char` array having 5 characters. This is bad and you should never do this.

Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];
c = "C programming";  // Error! array type is not assignable.
```

Dr. Uma

Read String from the user

Use the `scanf()` function to read a string.

The `scanf()` function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example 1: scanf() to read a string

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Example 3: Passing string to a Function

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);     // Passing string to a function.
    return 0;
}
void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

Dr. Uma

Strings and Pointers

Similar like arrays, string names are "decayed" to pointers.

Example 4: Strings and Pointers

```
#include <stdio.h>

int main(void) {
 char name[] = "Harry Potter";

 printf("%c", *name);    // Output: H
 printf("%c", *(name+1)); // Output: a
 printf("%c", *(name+7)); // Output: o

 char *namePtr;

 namePtr = name;
 printf("%c", *namePtr);    // Output: H
 printf("%c", *(namePtr+1)); // Output: a
 printf("%c", *(namePtr+7)); // Output: o
}
```

Even though `Dennis Ritchie` was entered in the above program, only `"Dennis"` was stored in the `name` string. It's because there was a space after `Dennis`.

How to read a line of text?

Use the `fgets()` function to read a line of string. And, you can use `puts()` to display the string.

fputs() & fgets()
write read string from stream

Example 2: fgets() and puts()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);  // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Dr. Uma

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used `fgets()` function to read a string from the user.

```
fgets(name, sizeof(name), stdlin); // read string
```

The `sizeof(name)` results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the `name` string.

To print the string, we have used `puts(name);`.

Note: The `gets()` function can also be to take input from the user. However, it is removed from the C standard.

It's because `gets()` allows you to input any length of characters. Hence, there might be a buffer overflow.

Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays.

C gets() and puts() functions

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

C gets() function

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration:

1. **char**[] gets(**char**[]);

Reading string using gets()

```
1. #include<stdio.h>
2. void main ()
3. {
4.     char s[30];
5.     printf("Enter the string? ");
6.     gets(s);
7.     printf("You entered %s",s);
8. }
```

Output

```
Enter the string?
Hi welcome at SRMIST
You entered Hi welcome at SRMIST
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read. Consider the following example.

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

```
1. void main()
2. {
3.    char str[20];
4.    printf("Enter the string? ");
5.    fgets(str, 20, stdin);
6.    printf("%s", str);
7. }
```

Output

C puts() function

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. **Declaration**

```
1. int puts(char[])
```

Let's see an example to read a string using gets() and print it on the console using puts().

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4. char name[50];
5. printf("Enter your name: ");
6. gets(name); //reads string from user
7. printf("Your name is: ");
8. puts(name);  //displays string
9. return 0;
10.}
```

Output:

getchar() function

Read a single character using the getchar() function

Let's consider a program to take a single using the getchar() function in C.

Program.c

```
#include <stdio.h>

void main()
{
   char c;
   printf ("\n Enter a character \n");
   c = getchar(); // get a single character
   printf(" You have passed ");
   putchar(c); // print a single character using putchar

   return 0;
}
```

Output

As we can see in the above program, it takes a single character at the run time from the user using the getchar() function. After getting the character, it prints the letter through the putchar() function.

putchar() function in C

The **putchar(int char)** method in C is used to write a character, of unsigned char type, to stdout. This character is passed as the parameter to this method.

Syntax:

```
int putchar(int char)
```

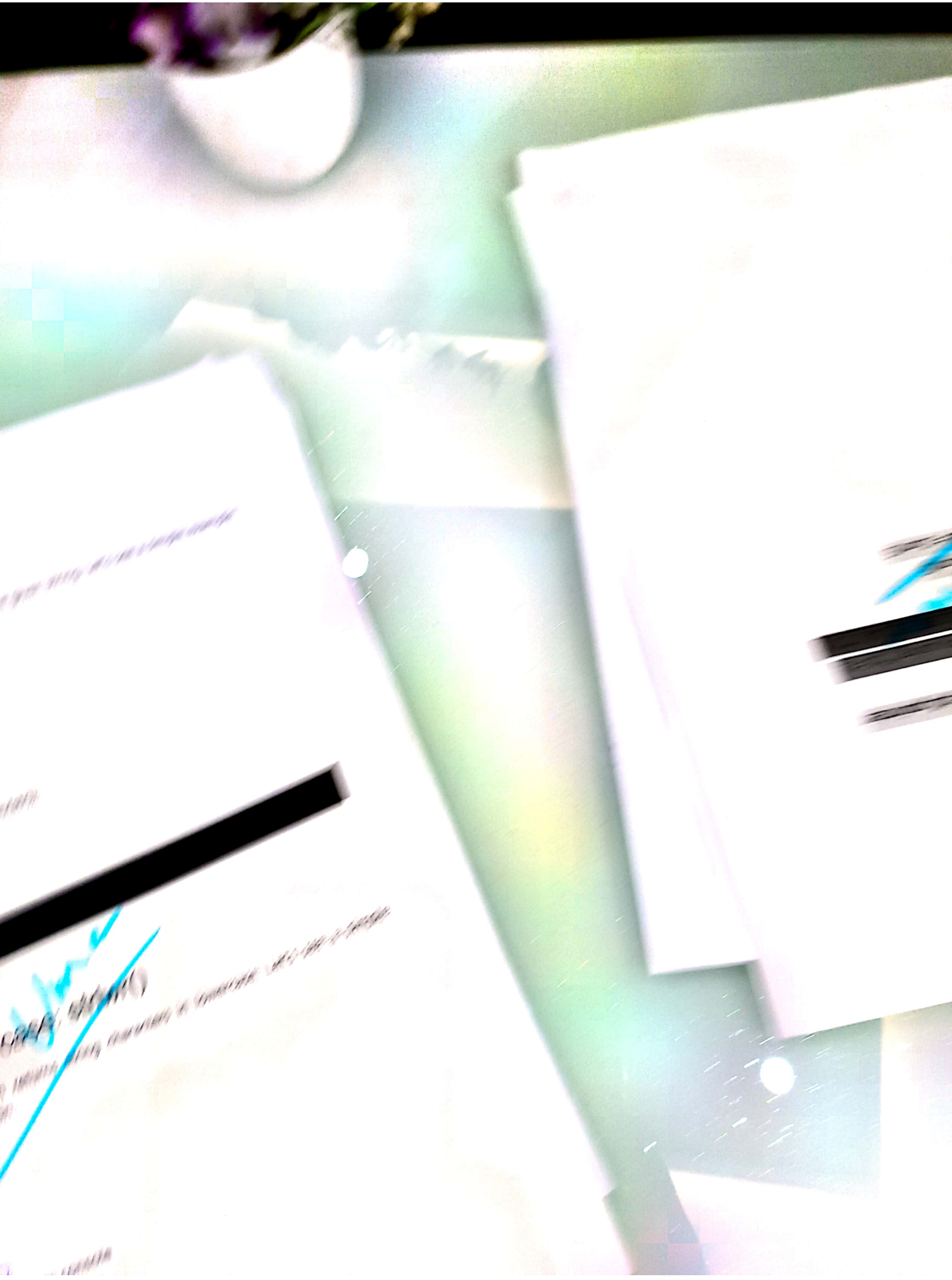
Parameters: This method accepts a mandatory parameter **char** which is the character to be written to stdout.

Return Value: This function returns the character written on the stdout as an unsigned char. It also returns EOF when some error occurs.

```c
#include <stdio.h>

int main()
{

   // Get the character to be written
   char ch = '1';

   // Write the Character to stdout
   for (ch = '1'; ch <= '9'; ch++)
      putchar(ch);

   return (0);
}
```

Output
:

```c
#include <stdio.h>

int main()
{

   // Get the character to be written
   char ch = '1';

   // Write the Character to stdout
   for (ch = '1'; ch <= '9'; ch++)
      putchar(ch);

   return (0);
}
```



```
9.  return 0;
10. }
```

Output:

C String Uppercase: strupr()

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

```
1.  #include<stdio.h>
2.  #include <string.h>
3.  int main(){
4.    char str[20];
5.    printf("Enter string: ");
6.    gets(str);//reads string from console
7.    printf("String is: %s",str);
8.    printf("\nUpper String is: %s",strupr(str));
9.   return 0;
10. }
```

Output:

Commonly Used String Functions

- strlen() - calculates the length of a string
- strcpy() - copies a string to another
- strcmp() - compares two strings
- strcat() - concatenates two strings

1.C strlen( )

The strlen() function calculates the length of a given string.

The `strlen()` function takes a string as an argument and returns its length. The returned value is of type `size_t` (the unsigned integer type).

It is defined in the <string.h> header file.

Example: C strlen() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[20]="Program";
    char b[20]={'P','r','o','g','r','a','m','\0'};

    // using the %zu format specifier to print size_t
    printf("Length of string a = %zu \n",strlen(a));
    printf("Length of string b = %zu \n",strlen(b));

    return 0;
}
```

```
Length of string a = 7
Length of string b = 7
```

Note that the `strlen()` function doesn't count the null character `\0` while calculating the length.

2.C strcat( )

In C programming, the strcat() function contcatenates (joins) two strings.

The function definition of `strcat()` is:

```
char *strcat(char *destination, const char *source)
```

It is defined in the `string.h` header file.

strcat() arguments

As you can see, the `strcat()` function takes two arguments:

destination - destination string

source - source string

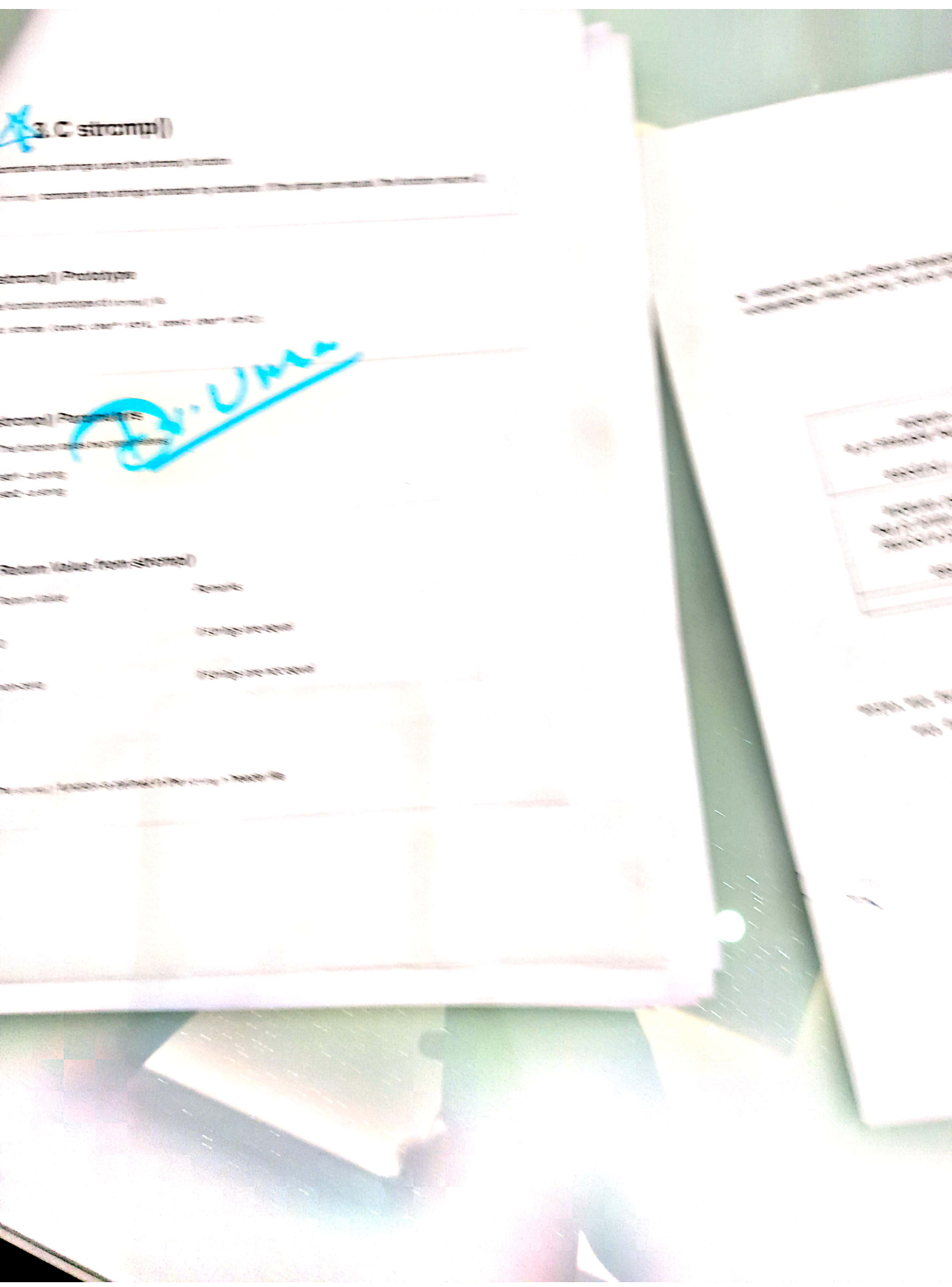
The `strcat()` function concatenates the `destination` string and the `source` string, and the result is stored in the `destination` string.

Example: C strcat() function

```
#include <stdio.h>
#include <string.h>
int main() {
  char str1[100] = "This is ", str2[] = "scm";

  // concatenates str1 and str2
  // the resultant string is stored in str1.
  strcat(str1, str2);

  puts(str1);
  puts(str2);

  return 0;
}
```

Output

```
This is scm
scm
```



Example: C strcmp() function

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

int main() {
  char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
  int result;

  // comparing strings str1 and str2
  result = strcmp(str1, str2);
  printf("strcmp(str1, str2) = %d\n", result);

  // comparing strings str1 and str3
  result = strcmp(str1, str3);
  printf("strcmp(str1, str3) = %d\n", result);

  return 0;
}
```

```
strcmp(str1, str2) = 1
strcmp(str1, str3) = 0
```

In the program,

- strings `str1` and `str2` are not equal. Hence, the result is a non-zero integer.
- strings `str1` and `str3` are equal. Hence, the result is 0.

4. C strcpy()

The strcpy() function in C programming to copy strings.

C strcpy()

The function prototype of `strcpy()` is:

```
char* strcpy(char* destination, const char* source);
```

- The `strcpy()` function copies the string pointed by `source` (including the null character) to the destination.
- The `strcpy()` function also returns the copied string.

The `strcpy()` function is defined in the `string.h` header file.

Example: C strcpy()

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

int main() {
  char str1[20] = "C programming";
  char str2[20];

  // copying str1 to str2
  strcpy(str2, str1);

  puts(str2); // C programming

  return 0;
}
```

```
C programming
```

Dr. Uma