Strings in C

In C we do not have a real string data type. We have to use different approach to store and handle strings. We use array of characters to store strings, with a convention that there is a special character (null) after the last character of the string in that array. In other words, a string in C language is actually a one-dimensional array of characters terminated by a null character – denoted by '\0'.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string *must* be one more than the number of characters in the word "Hello".

Example

The following are equivalent:

char greeting [6] = {'H', 'e', 'l', 'l', 'o', '\0'};

OR

char greeting [ ] = ''Hello'';

The representation of the above string in memory is as shown.

H	e	l	l	o	'\0'

You do not have to place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string. But if you initialize an array of characters you have to do it yourself.

Strings are declared in C in similar manner as arrays. Only difference is that, strings are always of char type.

Declaration

char string_name[size];

For example, char **str** [5];

str[0]	str[1]	str[2]	str[3]	str[4]

Strings can also be declared using pointer: char* p;

Initialization

In C, string can be initialized in a number of ways.

char c[ ] = ''abcd'';

OR char c[5] = ''abcd'';

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

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

OR const char* c = ''abcd'';

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	'\0'

String Input-Output

Let us look at the code right away.

```
1 #include <stdio.h>
2 int main() {
3     char name[20];
4     printf("Enter your name ");
5     scanf("%s", name);
6     printf("Hello %s\n", name);
7     return 0;
8 }
```

```
$
$ c99 Program-11-1.c
$ ./a.out
Enter your name Asokan Pichai
Hello Asokan
$
```

Figure 1: String input

From the code and the execution, you can see two things:

1. String variables read through `scanf()` can only take a word. It is because when white space is encountered, the `scanf()` function terminates. In the example, the second name is left in the buffer.
2. The name of the string variable is its address; there is no need for `&name`. This is true of arrays in general; if `p` is an array, `&p[0] == p`.

To write a string, we can use `printf()` function with format specifier "%s". We saw the use in the previous code, where we printed the name.

There are two complementary functions, `gets()` and `puts()` for reading strings from the console and printing them respectively. They are also part of the standard input-output library.

Example

```
#include <stdio.h>
int main() {
    char name[20];
    puts("Enter your name ");
    gets(name);
    printf("Hello %s\n", name);
    return 0;
}
```

Observations

- `scanf()` ends reading when a space is encountered; when input is "talent sprint", `scanf()` reads only "talent" and when input is "information technology", `scanf()` reads only "information"
- To solve this problem we can use `gets()` input function – but it is deprecated.
- Both `scanf()` and `gets()` add the necessary '\0' character at the end of the string.
- Both `scanf()` and `gets()` cannot avoid the problems when the number of characters input is greater than the size of the string.
- `puts()` adds a '\n' to the end of the string.
- `printf()` is needed for combining and formatting output


```
$ c99 Program-11-2.c
Program-11-2.c: In function ‘main’:
Program-11-2.c:6:3: warning: ‘gets’ is deprecated (declared at /usr/include/stdi
o.h:638) [-Wdeprecated-declarations]
   gets(name);
   ^
/tmp/cc5l4h55.o: In function `main':
Program-11-2.c:(.text+0x1a): warning: the `gets' function is dangerous and shoul
d not be used.
$ ./a.out
Enter your name
Asokan Pichai
Hello Asokan Pichai
$
```

Figure 2: Using gets()/puts()

We reproduce the relevant part of the `gets()` manual page to see why it is deprecated.

> Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

String Input

We take a closer look at the options for string input; we will use `scanf()` to read a string with a space in it (it will give us trouble!). And then we will use the recommended `fgets()`. Please remember to use `scanf()` or `fgets()` depending on whether white space is expected to be part of the string or not. In both cases ensure that the size of the variables is sufficient. When calculating sizes do not forget that we need one more for the '\0' character at the end. Thus "Hello" needs SIX character for storage and not five.

Examples

```
1 #include <stdio.h>
2 int main() {
3     char name[20];
4     char tmp[20];
```



```
5       puts("Enter your name ");
6       scanf("%s%s", name, tmp);
7       printf("Hello %s\n", name);
8       printf("What is this %s\n", tmp);
9       return 0;
10  }
```

Note that we use an extra variable to capture the data left in the buffer by `scanf`.

```
$ c99 Program-11-3.c
$ ./a.out
Enter your name
Asokan Pichai
Hello Asokan
What is this Pichai
$
```

Figure 3: scanf() trouble

Let us see how to use `fgets()`. The declaration is

char* fgets(char* string, **int** size, FILE* stream)

Ignore the third parameter for now. Just remember that to replace `gets()` with `fgets()` you need to use `stdin` as the third parameter.

```
1  #include <stdio.h>
2  int main() {
3       char name[20];
4       char tmp[20];
5       puts("Enter your name ");
6       fgets(name, 20, stdin);
7       printf("Hello <%s>\n", name);
8       return 0;
9  }
```

If you look at the output in Figure , you will notice a different problem: unlike `scanf()` and `gets()` the '\n' character is now part of the input and that needs to be handled! Let us write a function which will use `fgets()` inside but remove the '\n' character.

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



```
$
$ c99 Program-11-4.c
$ ./a.out
Enter your name
Asokan Pichai
Hello <Asokan Pichai
>
$
```

Figure 4: Using fgets()

```
2  #include <string.h>
3
4  char* getString(char* string, int size);
5
6  int main() {
7    char name[41];
8    printf("Enter name: ");
9    getString(name, 40);
10   printf("<%s>\n", name);
11   return 0;
12 }
13
14 char* getString(char* s, int n) {
15   fgets(s, n, stdin);
16   int linefeedPosition = strlen(s) - 1;
17   s[linefeedPosition] = '\0';
18   return s;
19 }
```

String Handling Functions

We can see the need to do the following:

- Finding the length of a string.
- Comparing two strings.


```
$
$ c99 Function-11.1.c
$ ./a.out
Enter name: Asokan Pichai
<Asokan Pichai>
$
```

Figure 5: getString() using fgets()

- Copying one string to another.
- Joining two strings.

There are functions defined in <*string.h* > header file to do these operations (and a few others). Let us look at these string handling functions.

strlen() Returns the length of its string argument. As its declaration shows, it does *not* modify the argument.

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

strcpy() Copies the contents of the second argument to the first argument. As the declaration shows, the source is unmodified while of course destination is modified. Returns a pointer to the destination string.

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

strcat() Concatenates – joins – two strings. It takes two arguments and appends contents of the second string to the first. Returns a pointer to the concatenated string.

- char* strcat (char* destination , const char* source);

strcmp() Compares two strings and returns value 0, if the two strings are equal. Returns a +ve integer if the first string is greater than the second. Returns -ve integer if first string is less than second. It does not modify either argument. [1]

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

[1]The comparison is based on lexical order. That is if the strings are sorted in dictionary order (that is characters with lower ASCII value come earlier than characters with greater ASCII value), and if str1 is earlier it returns a negative integer and if str1 is later it returns a positive integer.

Example

In this example we have used our own getString() function.

```
#include <stdio.h>
#include <string.h>
char* getString(char* string, int size);

int main() {
    char s[21];
    char t[21];
    char st[41];

    puts("Enter the first string");
    getString(s, 20);
    puts("Enter the second string");
    getString(t, 20);

    printf("Length of first is %d and second is %d\n",
            strlen(s), strlen(t));

    puts("Copying first to third");
    strcpy(st, s);
    puts("So third is now:");
    puts(st);
    puts("Appending second to third");
    strcat(st, t);
    puts("So third is now:");
    puts(st);

    int compare = strcmp(s, t);
    if (compare == 0) {
        puts("First and second are same");
    } else if (compare < 0) {
        printf("%s < %s\n", s, t);
    } else {
        printf("%s > %s\n", s, t);
```



```
34      }
35      return 0;
36  }
37
38  char* getString(char* s, int n) {
39      fgets(s, n, stdin);
40      int linefeedPosition = strlen(s) - 1;
41      s[linefeedPosition] = '\0';
42      return s;
43  }
```

```
$
$ c99 Program-11-5.c
$ ./a.out
Enter the first string
asokan pichai
Enter the second string
TalentSprint
Length of first is 13 and second is 12
Copying first to third
So third is now:
asokan pichai
Appending second to third
So third is now:
asokan pichaiTalentSprint
asokan pichai > TalentSprint
$
```

Figure 6: String functions

Strings and Pointers

Pointers and strings are closely related. We can treat them almost synonymously. Thus arrays of strings are easily dealt with as arrays of pointers.

Example

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



```
2  int main() {
3      char* langs[] = {"Pascal", "C", "C++", "Java"};
4      char* p;
5      for (int i = 0; i < 4; i++) {
6          p = langs[i];
7          printf("%10s is at %u\n", langs[i], p);
8      }
9      puts("--------------------");
10     char** q = langs;
11     for (int i = 0; i < 4; i++) {
12         printf("%10s is at %u\n", langs[i], *q++);
13     }
14     return 0;
15 }
```

```
$
$ c99 Program-11-6.c
$ ./a.out
    Pascal is at 4196004
         C is at 4196011
       C++ is at 4196013
      Java is at 4196017
--------------------
    Pascal is at 4196004
         C is at 4196011
       C++ is at 4196013
      Java is at 4196017
$
```

Figure 7: Addresses and Contents

Important Points

- All the string functions assume that sufficient memory is available. For example, if you say `strcat`(α, β), it is assumed that α was originally declared with a size $\geq$ `strlen`(α) + `strlen`(β) + 1.
- When pointers are used, its vital to ensure that overlapping strings are handled properly.

Arrays and Pointers

What we said about strings and pointers holds for arrays and pointers too. We can say arrays and pointers are synonymous in terms of how they access memory. But, the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed.

```
#include <stdio.h>
int main(){
    char message[5] = { 'H', 'e', 'l', 'p', 0 };
    int  prime10[] = { 2, 3, 5, 7};
    for(int i = 0; i < 4; ++i) {
        printf("Address of message[%d] is = %x  ", i, &message[i]);
        printf("prime10[%d] is = %x\n", i, &prime10[i]);
    }
    return 0;
}
```

```
$
$ c99 Program-11-7.c
$ ./a.out
Address of message[0] is = 26526850  prime10[0] is = 26526840
Address of message[1] is = 26526851  prime10[1] is = 26526844
Address of message[2] is = 26526852  prime10[2] is = 26526848
Address of message[3] is = 26526853  prime10[3] is = 2652684c
$
```

Figure 8: Array element addresses

Notice the equal difference between consecutive elements of the arrays. As expected it is one byte between successive elements of a character array and 4 bytes for an integer array.

Note: You will get different starting addresses for the arrays.

Pointer Arithmetic

Consider an array: **int** arr [4];

The name of the array is synonymous with the address of the first element of the array. Here,

address of first element of an array is `&arr[0]`.

`&arr[0]` ≡ `arr` and `arr[0]` ≡ to `*arr`).

`&arr[1]` ≡ `(arr + 1)` and `arr[1]` ≡ to `*(arr + 1)`.

`&arr[2]` ≡ `(arr + 2)` and `arr[2]` ≡ to `*(arr + 2)`.

`&arr[3]` ≡ `(arr + 3)` and `arr[3]` ≡ to `*(arr + 3)`.

Note

Now the most important point to remember is that adding 1 to a pointer makes it point to the next element; in this case it adds 4 bytes to the address.

The next program will help us understand this point well.

```
#include <stdio.h>
int main() {
    char msg[] = "Hello";
    int  fib[] = { 1, 1, 2, 3, 5, 8};

    for (char* p = msg; *p != '\0'; p++) {
        printf("Address in p is %u and it contains %c\n", p, *p);
    }
    printf("\n");
    int* p = fib;
    for (int i = 0; i < 6; i++, p++) {
        printf("Address in p is %u and it contains %d\n", p, *p);
    }
    return 0;
}
```

Dynamic Memory Allocation

We noted that there should be sufficient memory allocated for holding the data, when pointers are used. When we declare an array variable, its size is allocated at compile time and cannot be changed. But using pointers and dynamic memory allocation, we can handle this problem.

There are 4 library functions under *<stdlib.h >* for dynamic memory allocation.


```
$
$ c99 Program-11-8.c
$ ./a.out
Address in p is 617204400 and it contains H
Address in p is 617204401 and it contains e
Address in p is 617204402 and it contains l
Address in p is 617204403 and it contains l
Address in p is 617204404 and it contains o

Address in p is 617204368 and it contains 1
Address in p is 617204372 and it contains 1
Address in p is 617204376 and it contains 2
Address in p is 617204380 and it contains 3
Address in p is 617204384 and it contains 5
Address in p is 617204388 and it contains 8
$
```

Figure 9: Pointer increment

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

malloc() and free()

The name malloc stands for "memory allocation". The function `malloc()` reserves a block of memory of specified size and returns a pointer of type void which can be cast into pointer of any form. Its syntax is

```
ptr = (cast-type*) malloc(N);
```

Here, ptr is pointer of cast-type. The `malloc()` function returns a pointer to an area of memory whose size is **N**. If the memory available is insufficient, allocation fails and malloc returns *NULL* pointer.

```
int* ptr = (int*) malloc(100 * sizeof (int ));
```

This statement declares a new pointer `ptr`, and allocates memory to store will 100 integer (400 bytes) and stores the address of the first byte of that block in `ptr`.

Please note that the memory is *NOT* initialized.

We need to call the function `free()` to release the memory which was allocated.

Let us write a program to sum the numbers entered by a user. If we use arrays to store the numbers, we have to assume that the user does not enter more than a given set of numbers.

We can remove that restriction, by allocating memory dynamically using `malloc()` function.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int* ptr = (int*) malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for (int i = 0; i < n; ++i) {
        scanf("%d",ptr+i);
    }
    int total = 0;
    for (int i = 0; i < n; ++i) {
        total += *(ptr + i);
    }
    printf("Sum = %d\n", total);
    free(ptr);
    return 0;
}
```

calloc()

The name calloc stands for "contiguous allocation". The major difference between `malloc()` and `calloc()` is that, `malloc()` allocates a single block of memory whereas `calloc()` allocates multiple blocks of memory each of the specified size and sets all the allocated memory to zero.


```
$
$ c99 Program-11-9.c
$ ./a.out
Enter number of elements: 11
Enter elements of array: 1 3 5 7 9 11 12 14 16 18 20
Sum = 116
$
```

Figure 10: malloc

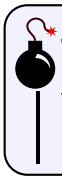
`ptr = (cast-type*) calloc(n, element-size);` This statement will allocate contiguous space in memory for an array of n elements. For example:

ptr = (**float***) calloc (25, sizeof (**float**)); allocates contiguous space in memory for an array of 25 elements each of size of **float**, namely 4 bytes.

realloc()

If the previously allocated memory is insufficient or too much, you can change using `realloc()`.

`ptr = realloc(ptr, newsize);` Here, ptr is reallocated with size of newsize.



WARNING

The pointer given as argument to free() and realloc() must have the address as returned by an earlier malloc(), calloc() or realloc(). If that is not so, the results are undefined.

