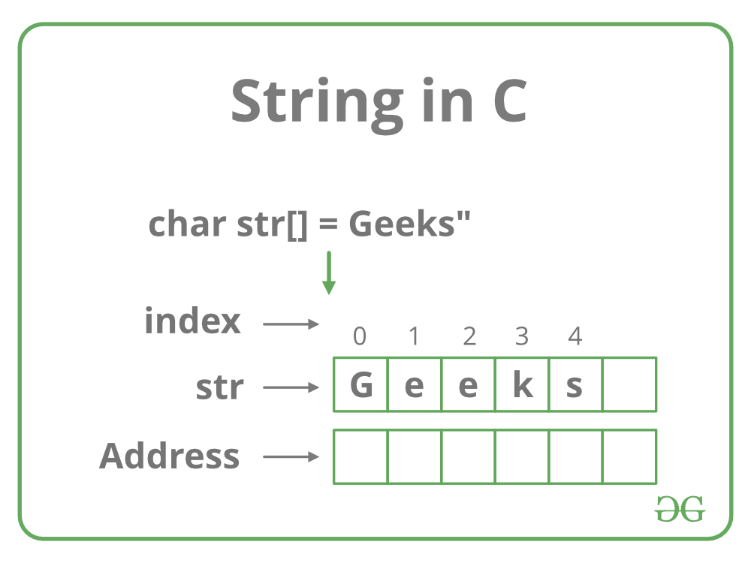
**Strings in C**

Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character ‘\0’.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20191113124735/CString.png)

**Declaration of strings**: Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string.

char str\_name[size];

In the above syntax str\_name is any name given to the string variable and size is used define the length of the string, i.e the number of characters strings will store. Please keep in mind that there is an extra terminating character which is the Null character (‘\0’) used to indicate termination of string which differs strings from normal character arrays.

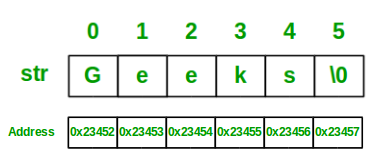
**Initializing a String**: A string can be initialized in different ways. We will explain this with the help of an example. Below is an example to declare a string with name as str and initialize it with “GeeksforGeeks”.

1. char str[] = "GeeksforGeeks";

2. char str[50] = "GeeksforGeeks";

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

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

Below is the memory representation of a string “Geeks”.  


Let us now look at a sample program to get a clear understanding of declaring and initializing a string in C and also how to print a string.

|  |
| --- |
| // C program to illustrate strings    #include<stdio.h>    int main()  {      // declare and initialize string      char str[] = "Geeks";        // print string      printf("%s",str);        return 0;  } |

Output:

Geeks

We can see in the above program that strings can be printed using a normal printf statements just like we print any other variable. Unlike arrays we do not need to print a string, character by character. The C language does not provide an inbuilt data type for strings but it has an access specifier “**%s**” which can be used to directly print and read strings.

**Below is a sample program to read a string from user**:

|  |
| --- |
| // C program to read strings    #include<stdio.h>    int main()  {      // declaring string      char str[50];        // reading string      scanf("%s",str);        // print string      printf("%s",str);        return 0;  } |

You can see in the above program that string can also be read using a single scanf statement. Also you might be thinking that why we have not used the ‘&’ sign with string name ‘str’ in scanf statement! To understand this you will have to recall your knowledge of scanf. We know that the ‘&’ sign is used to provide the address of the variable to the scanf() function to store the value read in memory. As str[] is a character array so using str without braces ‘[‘ and ‘]’ will give the base address of this string. That’s why we have not used ‘&’ in this case as we are already providing the base address of the string to scanf.

**Passing strings to function**: As strings are character arrays, so we can pass strings to function in a same way we pass an array to a function. Below is a sample program to do this:

|  |
| --- |
| // C program to illustrate how to  // pass string to functions  #include<stdio.h>    void printStr(char str[])  {      printf("String is : %s",str);  }    int main()  {      // declare and initialize string      char str[] = "GeeksforGeeks";        // print string by passing string      // to a different function      printStr(str);      return 0;  } |

Output:

String is : GeeksforGeeks

### **String Handling Functions:**

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| strcat() | It is used to concatenate(combine) two strings |
| strlen() | It is used to show length of a string |
| strrev() | It is used to show reverse of a string |
| strcpy() | Copies one string into another |
| strcmp() | It is used to compare two string |

#### **strcat() function**

strcat("hello", "world");

strcat() function will add the string **"world"** to **"hello"** i.e it will ouput helloworld.

#### **strlen() function**

strlen() function will return the length of the string passed to it.

int j;

j = strlen("studytonight");

printf("%d",j);

12

#### **strcmp() function**

strcmp() function will return the ASCII difference between first unmatching character of two strings.

int j;

j = strcmp("study", "tonight");

printf("%d",j);

-1

#### **strcpy() function**

It copies the second string argument to the first string argument.

#include<stdio.h>

#include<string.h>

int main()

{

char s1[50];

char s2[50];

strcpy(s1, "StudyTonight"); //copies "studytonight" to string s1

strcpy(s2, s1); //copies string s1 to string s2

printf("%s\n", s2);

return(0);

}

#### **strrev() function**

It is used to reverse the given string expression.

#include<stdio.h>

int main()

{

char s1[50];

printf("Enter your string: ");

gets(s1);

printf("\nYour reverse string is: %s",strrev(s1));

return(0);

}

Enter your string: studytonight Your reverse string is: thginotyduts

Strings

I mentioned early on that strings and words couldn't be stored in a normal variable. However, with an array of characters, we have that power. The difference between just an array of characters and a string in C is the addition of a NULL character (**\0**) at the end. NULL indicates where the end of the string is located.

|  |  |  |
| --- | --- | --- |
| http://archive.oreilly.com/oreillyschool/courses/c/images/word.jpg |  | This is just a character array |
| http://archive.oreilly.com/oreillyschool/courses/c/images/word2.jpg |  | This is a string |

**Declaring Strings:**

Strings can be declared using the same method used to specify values in a character array or we can use a shortcut by including a string in double quotes. When we use double quotes, the NULL character is automatically added.

OBSERVE:

char string1[]={'y','o','\0'};

char string2[]="hello";

char string3[6];

The first string ends up being a three-character array with the string "yo" stored inside (**\0** means NULL). Because it uses double quotes, **string2** automatically includes the **\0** at the end. **string3** is essentially just declared as an array of characters. In order for it to be a string, we have to remember to include the **\0** at the end.

Using Strings

Let's start using our strings!

/\* string\_printer.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char string1[]={'y','o','\0'};

char string2[]="hello";

char string3[6];

strcpy(string3,"there");

printf("%s ",string1);

printf("%s ",string2);

printf("%s\n",string3);

return 1;

}

Save it as **string\_printer.c**, and compile and run it to see it in action.

Output from string\_printer.exe

yo hello there

When using strings, the first thing we need to do is include an additional header file named **string.h**. This allows us to use many of the functions designed to work with strings. The strcpy() statement is a way to assign a value to a string. Unlike with normal variables, we can only use the assignment operator (**=**) when declaring a string. To set a value later, we must use something like strcpy(). strcpy() can also be used to copy one string into another. In this case, we set **string3** equal to "there." Remember to keep the size of the string in mind so that it does not exceed the size of the array when a NULL character is added to the end.

It's simple to print out strings using a **%s**. That way we don't have to worry about where the end of the string is.

|  |  |
| --- | --- |
| Note | **string.h** isn't necessary on all platforms, but it's better to include it than try to remember which ones it's needed for. |

For the rest of this lesson I'll briefly cover some of the functions most commonly used with strings. **Compile and test** all of the following string examples to get a feel for them. Play around a bit and see what you can come up with.

**strcat:**

The strcat() function, as with all of the functions I'll go over that start with "str," requires that you include **string.h**.

Type the following into CodeRunner:

/\* concatenator.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char first[40] = "The first string ";

char second[] = "and the second string.\n";

strcat(first,second);

printf("%s",first);

return 1;

}

The strcat() function appends the second string to the end of the first string. (They don't have to be called **first** and **second**, but it makes it easier to illustrate.) What about the NULL character at the end of the first string? Well, luckily for us, strcat() handles that by putting the first character of the second string where the NULL used to be. The result is this:

Output from concatenator.exe

The first string and the second string.

We could also put strcat() directly in the print statement:

Observe the following:

printf("%s",strcat(first,second));

This is because strcat() returns a pointer to the first string, which is exactly the same thing that the name **first** refers to.

**strcmp:**

Use the strcmp() function to compare two strings. It returns an integer value based on the result of the comparison. It returns less than 0 if the first string is less than the second, 0 if they are equal, and greater than 0 if the first is greater than the second. Wait a second. Less than and greater than? I thought we were talking about characters here. We are, but remember, characters in C are represented by numbers as well!

Type the following into CodeRunner below:

/\* comparator.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char one[] = "abcd";

char two[] = "abcz";

void compare(char \*str1, char \*str2);

compare(one, one);

compare(one, two);

compare(two, one);

return 1;

}

void compare(char \*str1, char \*str2){

int value;

value = strcmp(str1,str2);

if(value < 0){

printf("%s is less than %s\n",str1,str2);

}else if(value == 0)

printf("%s is equal to %s\n",str1,str2);

else if(value > 0){

printf("%s is greater than %s\n",str1,str2);

}

}

Save this one as **comparator.c**. Try it out to get a feel for what it does. Here I went an extra step and wrote another function to show all the different outcomes. The **void** type is used when we don't need the function to return anything.

**strlen:**

The strlen() function is used to find the length of a string. It's useful when you need to traverse a string like an array, and you don't know how long it is. It reads through the string, counting characters, until it sees the NULL character at the end. As a result, it's very important that NULL character is there.

Observe the following:

/\* how\_long.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char str[] = "Mary had a little lamb.";

int length;

length = strlen(str);

printf("The string is %i characters long.\n",length);

return 1;

}

The NULL character is not counted in the length of the string. The result looks like this:

Observe the following:

The string is 23 characters long.

**sprintf:**

The sprintf() function works almost exactly the same as printf(), except that you have to give it the name of your string first, and it saves its results in a string instead of printing to standard output (STDOUT).

Type the following into CodeRunner below:

/\* string\_maker.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char str[50];

int bob = 3;

sprintf(str,"This is my new string and bob equals %i\n",bob);

printf("%s",str);

return 1;

}

Save this as **string\_maker.c** before compiling and running it.

|  |  |
| --- | --- |
| Note | When working with strings, be very careful not to exceed the length of the string (one less than the size of the array). Writing into memory past the end of a string will commonly cause a lot of really weird errors. |

**sscanf:**

Just like we have sprintf(), we've also got sscanf(). This time, instead of reading from standard input (STDIN) it reads from, you guessed it, a string!

Type the following into CodeRunner below:

/\* string\_scanner.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char str[] = "a b c";

char one,two,three;

sscanf(str,"%c %c %c",&one,&two,&three);

printf("the characters are: %c %c %c\n",three,two,one);

return 1;

}

Save as **string\_scanner.c**, compile, and run. As you can see, the format is exactly the same as in scanf(), except that you have to tell it the name of your input string first.

**gets:**

The gets() function reads from STDIN, just like scanf(). However, the difference is that gets() keeps reading characters until a newline or end of file (EOF). It then stores the characters into a string. gets() will automatically append the NULL character to the end of the string as well.

Whenever you compile a program that uses gets(), you'll get an error message saying that using gets() is dangerous. Ignore this warning for now, but if you want it to go away you'll need to read the link below about programming strings more securely.

|  |  |
| --- | --- |
| Note | The EOF (end of file) character is found at the end of a file (as we'll see later) or an input stream. |

Type the following into CodeRunner below:

/\* string\_getter.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char str[100];

gets(str);

printf("%s",str);

return 1;

}

Save it as **string\_getter.c** and try it out to see it in action.

**Secure Strings**

Functions like strcpy, strcat, sprintf, and gets don't check the size of the string they're modifying. It's really easy to write past the end of your string. When this happens, your program might crash, or it might start executing code you didn't write. This problem is known as a *buffer overflow*, and is the reason many programs and operating systems have security problems.

The solution to this problem is to be extremely careful with the size of the strings. Lucky for us, there are four replacement functions that help do this: strncpy, strncat, snprintf, and fgets.

strncpy and strncat work very similarly. They require a number as an extra input. Only the first few characters of the second string, up to that number, are considered. The problem, is that you may loose the NULL character at the end of the first string, so you have to put one back.

Type the following into CodeRunner below:

/\* secure.c \*/

#include <stdio.h>

#include <string.h>

int main(){

char str[100];

char str2[] = "hello there";

int a=5;

strncpy(str,str2,a);

str[a] = '\0';

printf("%s\n",str);

return 1;

}

The result of the above sample follows. As you can see, only the first 5 characters of str2 are in str. Since we're copying the first 5 characters, we want a NULL character at the sixth spot which just happens to be the 5th index.

OBSERVE:

hello

strncmp works pretty close to the same way, except that the number limits the number of characters of the first string that are compared to the second string.

The function that is the most different from it's counterpart is fgets. First, fgets was designed to read from files as well as STDIN, so we have to specify that we want to use standard input as well as the maximum number of characters to read. It will read in up to one less than the number you specify (it saves room for the NULL character it adds) or until it reaches a newline. Unlike gets that doesn't keep a trailing newline as part of the string, fgets does. Another difference, is that gets also considers an EOF the end of a string, while fgets considers an EOF to be an error. If an error occurs, fgets returns a NULL pointer.

Here's an example that reads in the first ten characters:

Type the following into CodeRunner:

#include <stdio.h>

int main(){

char str[20];

int i=10;

if(fgets(str,i,stdin) == NULL){

printf("read failed.\n");

}

printf("%s\n",str);

return 1;

}

Try running this program, typing in a something like "Now is the time for all good C programmers."

OBSERVE:

Now is the time for all good C programmers

Now is th

cold:~$

The program only read in the first ten characters: **Now is th**.

These functions alone are not enough to ensure your program is safe from buffer overflows, but using them is definately a good idea