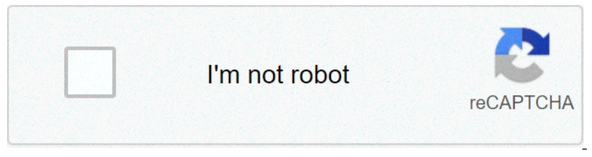

Continue

C declare string

String is an array of characters. In this guide, we learn how to declare strings, how to work with strings in C programming and how to use the pre-defined string handling functions. We will see how to compare two strings, concatenate strings, copy one string to another & perform various string manipulation operations. We can perform such operations using the pre-defined functions of “string.h” header file. In order to use these string functions you must include string.h file in your C program. String Declaration Method 1: char address[]={'T', 'E', 'X', 'A', 'S', '\0'}; Method 2: The above string can also be defined as – char address[]="TEXAS"; In the above declaration NULL character (\0) will automatically be inserted at the end of the string. What is NULL Char “\0”? '\0' represents the end of the string. It is also referred as String terminator & Null Character. String I/O in C programming Read & write Strings in C using Printf() and Scanf() functions #include #include int main() { /* String Declaration*/ char nickname[20]; printf("Enter your Nick name:"); /* I am reading the input string and storing it in nickname * Array name alone works as a base address of array so * we can use nickname instead of &nickname here */ scanf("%s", nickname); /*Displaying String*/ printf("%s",nickname); return 0; } Output: Enter your Nick name:Negan Negan Note: %s format specifier is used for strings input/output Read & Write Strings in C using gets() and puts() functions #include #include int main() { /* String Declaration*/ char nickname[20]; /* Console display using puts */ puts("Enter your Nick name:"); /*Input using gets*/ gets(nickname); puts(nickname); return 0; } C – String functions C String function – strlen Syntax: size_t strlen(const char *str) size_t represents unsigned short It returns the length of the string without including end character (terminating char '\0'). Example of strlen: #include #include int main() { char str1[20] = "BeginnersBook"; printf("Length of string str1: %d", strlen(str1)); return 0; } Output: Length of string str1: 13 strlen vs sizeof strlen returns you the length of the string stored in array, however sizeof returns the total allocated size assigned to the array. So if I consider the above example again then the following statements would return the below values. strlen(str1) returned value 13. sizeof(str1) would return value 20 as the array size is 20 (see the first statement in main function). C String function – strnlen Syntax: size_t strnlen(const char *str, size_t maxlen) size_t represents unsigned short It returns length of the string if it is less than the value specified for maxlen (maximum length) otherwise it returns maxlen value. Example of strnlen: #include #include int main() { char str1[20] = "BeginnersBook"; printf("Length of string str1 when maxlen is 30: %d", strnlen(str1, 30)); printf("Length of string str1 when maxlen is 10: %d", strnlen(str1, 10)); return 0; } Output: Length of string str1 when maxlen is 30: 13 Length of string str1 when maxlen is 10: 10 Have you noticed the output of second printf statement, even though the string length was 13 it returned only 10 because the maxlen was 10. C String function – strcmp int strcmp(const char *str1, const char *str2) It compares the two strings and returns an integer value. If both the strings are same (equal) then this function would return 0 otherwise it may return a negative or positive value based on the comparison. If string1 < string2 OR string1 is a substring of string2 then it would result in a negative value. If string1 > string2 then it would return positive value. If string1 == string2 then you would get 0(zero) when you use this function for compare strings. Example of strcmp: #include #include int main() { char s1[20] = "BeginnersBook"; char s2[20] = "BeginnersBook.COM"; if (strcmp(s1, s2) ==0) { printf("string 1 and string 2 are equal"); }else { printf("string 1 and 2 are different"); } return 0; } Output: string 1 and 2 are different C String function – strncmp int strncmp(const char *str1, const char *str2, size_t n) size_t is for unassigned short It compares both the string till n characters or in other words it compares first n characters of both the strings. Example of strncmp: #include #include int main() { char s1[20] = "BeginnersBook"; char s2[20] = "BeginnersBook.COM"; /* below it is comparing first 8 characters of s1 and s2*/ if (strncmp(s1, s2, 8) ==0) { printf("string 1 and string 2 are equal"); }else { printf("string 1 and 2 are different"); } return 0; } Output: string1 and string 2 are equal C String function – strcat char *strcat(char *str1, char *str2) It concatenates two strings and returns the concatenated string. Example of strcat: #include #include int main() { char s1[10] = "Hello"; char s2[10] = "World"; strcat(s1,s2); printf("Output string after concatenation: %s", s1); return 0; } Output: Output string after concatenation: HelloWorld C String function – strncat char *strncat(char *str1, char *str2, int n) It concatenates n characters of str2 to string str1. A terminator char ('\0') will always be appended at the end of the concatenated string. Example of strncat: #include #include int main() { char s1[10] = "Hello"; char s2[10] = "World"; strncat(s1,s2, 3); printf("Concatenation using strncat: %s", s1); return 0; } Output: Concatenation using strncat: HelloWor C String function – strcpy char *strcpy(char *str1, char *str2) It copies the string str2 into string str1, including the end character (terminator char '\0'). Example of strcpy: #include #include int main() { char s1[30] = "string 1"; char s2[30] = "string 2 : I'm gonna copied into s1"; /* this function has copied s2 into s1*/ strcpy(s1,s2); printf("String s1 is: %s", s1); return 0; } Output: String s1 is: string 2: I'm gonna copied into s1 C String function – strncpy char *strncpy(char *str1, char *str2, size_t n) size_t is unassigned short and n is a number. Case1: If length of str2 > n then it just copies first n characters of str2 into str1. Case2: If length of str2 < n then it copies all the characters of str2 into str1 and appends several terminator chars('\0') to accumulate the length of str1 to make it n. Example of strncpy: #include #include int main() { char first[30] = "string 1"; char second[30] = "string 2: I'm using strncpy now"; /* this function has copied first 10 chars of s2 into s1*/ strncpy(s1,s2, 12); printf("String s1 is: %s", s1); return 0; } Output: String s1 is: string 2: I'm C String function – strchr char *strchr(char *str, int ch) It searches string str for character ch (you may be wondering that in above definition I have given data type of ch as int, don't worry I didn't make any mistake it should be int only. The thing is when we give any character while using strchr then it internally gets converted into integer for better searching. Example of strchr: #include #include int main() { char mystr[30] = "I'm an example of function strchr"; printf ("%s", strchr(mystr, 'f')); return 0; } Output: f function strchr C String function – Strrchr char *strrchr(char *str, int ch) It is similar to the function strchr, the only difference is that it searches the string in reverse order, now you would have understood why we have extra r in strrchr, yes you guessed it correct, it is for reverse only. Now let's take the same above example: #include #include int main() { char mystr[30] = "I'm an example of function strchr"; printf ("%s", strrchr(mystr, 'f')); return 0; } Output: function strchr Why output is different than strchr? It is because it started searching from the end of the string and found the first 'f' in function instead of 'of'. C String function – strstr char *strstr(char *str, char *srch_term) It is similar to strchr, except that it searches for string srch_term instead of a single char. Example of strstr: #include #include int main() { char inputstr[70] = "String Function in C at BeginnersBook.COM"; printf ("Output string is: %s", strstr(inputstr, 'Begi')); return 0; } Output: Output string is: BeginnersBook.COM You can also use this function in place of strchr as you are allowed to give single char also in place of search_term string. From cppreference.com "s-char-sequence" (1) L"s-char-sequence" (2) u8"s-char-sequence" (3) (since C++11) u"s-char-sequence" (4) (since C++11) U"s-char-sequence" (5) (since C++11) prefix(optional) R"delimiter(raw_characters)delimiter" (6) (since C++11) s-char-sequence - A sequence of zero or more s-chars. An s-char is one of prefix - One of L, u8, u, U delimiter - A character sequence made of any source character but parentheses, backslash and spaces (can be empty, and at most 16 characters long) raw_characters - Any character sequence, except that it must not contain the closing sequence)delimiter" 1) Narrow multibyte string literal. The type of an unprefixed string literal is const char[N], where N is the size of the string in code units of the execution narrow encoding, including the null terminator. 2) Wide string literal. The type of a L"..." string literal is const wchar_t[N], where N is the size of the string in code units of the execution wide encoding, including the null terminator. 3) UTF-8 encoded string literal. The type of a u8"..." string literal is const char[N] (until C++20)const char8_t[N] (since C++20), where N is the size of the string in UTF-8 code units including the null terminator. 4) UTF-16 encoded string literal. The type of a u"..." string literal is const char16_t[N], where N is the size of the string in UTF-16 code units including the null terminator. 5) UTF-32 encoded string literal. The type of a U"..." string literal is const char32_t[N], where N is the size of the string in UTF-32 code units including the null terminator. 6) Raw string literal. Used to avoid escaping of any character. Anything between the delimiters becomes part of the string. prefix, if present, has the same meaning as described above. Each s-char initializes the corresponding element(s) in the string literal object. An s-char corresponds to more than one element if and only if it is represented by a sequence of more than one code units in the string literal's associated character encoding. If a character lacks representation in the associated character encoding, if the string literal is an ordinary string literal or wide string literal, it is conditionally-supported and an implementation-defined code unit sequence is encoded; otherwise (the string literal is UTF-encoded), the string literal is ill-formed. (since C++23) Each numeric escape sequence corresponds to a single element. If the value specified by the escape sequence fits within the unsigned version of the element type, the element has the specified value (possibly after conversion to the element type); otherwise (the specified value is out of range), the string literal is ill-formed. (since C++23) [edit] Concatenation String literals placed side-by-side are concatenated at translation phase 6 (after the preprocessor). That is, "Hello," " world!" yields the (single) string "Hello, world!". If the two strings have the same encoding prefix (or neither has one), the resulting string will have the same encoding prefix (or no prefix). If one of the strings has an encoding prefix and the other doesn't, the one that doesn't will be considered to have the same encoding prefix as the other. L"Δx = %" PRId16 // at phase 4, PRId16 expands to "d" // at phase 6, L"Δx = %" and "d" form L"Δx = %d" If a UTF-8 string literal and a wide string literal are side by side, the program is ill-formed. (since C++11) Any other combination of encoding prefixes may or may not be supported by the implementation. The result of such a concatenation is implementation-defined. (since C++11)(until C++23) Any other combination of encoding prefixes is ill-formed. (since C++23) [edit] Notes The null character ('\0', L'\0', char16_t(), etc) is always appended to the string literal: thus, a string literal "Hello" is a const char[6] holding the characters 'H', 'e', 'l', 'l', 'o', and '\0'. The encoding of narrow multibyte string literals (1) and wide string literals (2) is implementation-defined. For example, gcc selects them with the commandline options -fexec-charset and -fwide-exec-charset. String literals have static storage duration, and thus exist in memory for the life of the program. String literals can be used to initialize character arrays. If an array is initialized like char str[] = "foo";, str will contain a copy of the string "foo". Whether string literals can overlap and whether successive evaluations of a string-literal yield the same object is unspecified. That means that identical string literals may or may not compare equal when compared by pointer. bool b = "bar" == 3+"foobar" // could be true or false, implementation-defined Attempting to modify a string literal results in undefined behavior: they may be stored in read-only storage (such as .rodata) or combined with other string literals: const char* pc = "Hello"; char* p = const_cast(pc); p[0] = 'M'; // undefined behavior String literals are convertible and assignable to non-const char* or wchar_t* in order to be compatible with C, where string literals are of types char[N] and wchar_t[N]. Such implicit conversion is deprecated. (until C++11) String literals are not convertible or assignable to non-const CharT*. An explicit cast (e.g. const_cast) must be used if such conversion is wanted. (since C++11) A string literal is not necessarily a null-terminated character sequence: if a string literal has embedded null characters, it represents an array which contains more than one string. const char* p = "abc\0def"; // std::strlen(p) == 3, but the array has size 8 If a valid hex digit follows a hex escape in a string literal, it would fail to compile as an invalid escape sequence. String concatenation can be used as a workaround: //const char* p = "\xfff"; // error: hex escape sequence out of range const char* p = "\xff""f"; // OK: the literal is const char[3] holding {'\xff','f','\0'} Although mixed wide string literal concatenation is allowed in C++11, all known C++ compilers reject such concatenation, and its usage experience is unknown. As a result, allowance of mixed wide string literal concatenation is removed in C++23. [edit] Example #include char array1[] = "Foo" "bar"; // same as char array2[] = { 'F', 'o', 'o', 'b', 'a', 'r', '\0' }; const char* s1 = R"foo(Hello World)foo"; // same as const char* s2 = "Hello World"; // same as const char* s3 = "" "Hello" " World"; const wchar_t* s4 = L"ABC" L"DEF"; // ok, same as const wchar_t* s5 = L"ABCDEF"; const char32_t* s6 = U"GHI" "JKL"; // ok, same as const char32_t* s7 = U"GHIJKL"; const char16_t* s9 = "MN" u"OP" "QR"; // ok, same as const char16_t* sA = u"MNOPQR"; // const auto* sB = u"Mixed" U"Types"; // before C++23 may or may not be supported by // the implementation; ill-formed since C++23 const wchar_t* sC = LR"--(STUV)--"; // ok, raw string literal int main() { std::cout