6.6 — C-style strings

BY ALEX ON JULY 9TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In lesson **4.4b -- An introduction to std::string**, we defined a string as a collection of sequential characters, such as "Hello, world!". Strings are the primary way in which we work with text in C++, and std::string makes working with strings in C++ easy.

Modern C++ supports two different types of strings: std::string (as part of the standard library), and C-style strings (natively, as inherited from the C language). It turns out that std::string is implemented using C-style strings. In this lesson, we'll take a closer look at C-style strings.

C-style strings

A **C-style string** is simply an array of characters that uses a null terminator. A **null terminator** is a special character ('\0', ascii code 0) used to indicate the end of the string. More generically, A C-style string is called a **null-terminated string**.

To define a C-style string, simply declare a char array and initialize it with a string literal:

```
1 char myString[]{ "string" };
```

Although "string" only has 6 letters, C++ automatically adds a null terminator to the end of the string for us (we don't need to include it ourselves). Consequently, myString is actually an array of length 7!

We can see the evidence of this in the following program, which prints out the length of the string, and then the ASCII values of all of the characters:

```
1  #include <iostream>
2  #include <iterator> // for std::size
3
4  int main()
5  {
6      char myString[]{ "string" };
7      const int length{ static_cast<int>(std::size(myString)) };
8  //  const int length{ sizeof(myString) / sizeof(myString[0]) }; // use instead if not C++17 c
9  apable
10     std::cout << myString<< " has " << length << " characters.\n";
11
12     for (int index{ 0 }; index < length; ++index)
13         std::cout << static_cast<int>(myString[index]) << " ";
14
15     std::cout << '\n';
16
17     return 0;
   }
```

This produces the result:

```
string has 7 characters.
115 116 114 105 110 103 0
```

That 0 is the ASCII code of the null terminator that has been appended to the end of the string.

When declaring strings in this manner, it is a good idea to use [] and let the compiler calculate the length of the array. That way if you change the string later, you won't have to manually adjust the array length.

One important point to note is that C-style strings follow *all* the same rules as arrays. This means you can initialize the string upon creation, but you can not assign values to it using the assignment operator after that!

```
char myString[]{ "string" }; // ok
myString = "rope"; // not ok!
```

Since C-style strings are arrays, you can use the [] operator to change individual characters in the string:

```
#include <iostream>

int main()
{
    char myString[]{ "string" };
    myString[1] = 'p';
    std::cout << myString << '\n';

    return 0;
}
```

This program prints:

```
spring
```

When printing a C-style string, std::cout prints characters until it encounters the null terminator. If you accidentally overwrite the null terminator in a string (e.g. by assigning something to myString[6]), you'll not only get all the characters in the string, but std::cout will just keep printing everything in adjacent memory slots until it happens to hit a 0!

Note that it's fine if the array is larger than the string it contains:

```
#include <iostream>

int main()
{
    char name[20]{ "Alex" }; // only use 5 characters (4 letters + null terminator)
    std::cout << "My name is: " << name << '\n';

    return 0;
}
```

In this case, the string "Alex" will be printed, and std::cout will stop at the null terminator. The rest of the characters in the array are ignored.

C-style strings and std::cin

There are many cases where we don't know in advance how long our string is going to be. For example, consider the problem of writing a program where we need to ask the user to enter their name. How long is their name? We don't know until they enter it!

In this case, we can declare an array larger than we need:

```
#include <iostream>

int main()
{
    char name[255]; // declare array large enough to hold 255 characters
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "You entered: " << name << '\n';

    return 0;
```

```
}
```

In the above program, we've allocated an array of 255 characters to name, guessing that the user will not enter this many characters. Although this is commonly seen in C/C++ programming, it is poor programming practice, because nothing is stopping the user from entering more than 255 characters (either unintentionally, or maliciously).

The recommended way of reading C-style strings using `std::cin` is as follows:

```
#include <iostream>
#include <iterator> // for std::size

int main()
{
    char name[255]; // declare array large enough to hold 255 characters
    std::cout << "Enter your name: ";
    std::cin.getline(name, std::size(name));
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

This call to cin.getline() will read up to 254 characters into name (leaving room for the null terminator!). Any excess characters will be discarded. In this way, we guarantee that we will not overflow the array!

Manipulating C-style strings

C++ provides many functions to manipulate C-style strings as part of the <cstring> library. Here are a few of the most useful:

strcpy() allows you to copy a string to another string. More commonly, this is used to assign a value to a string:

```
#include <cstring>

int main()
{
    char source[]{ "Copy this!" };
    char dest[50];
    std::strcpy(dest, source);
    std::cout << dest << '\n'; // prints "Copy this!"

    return 0;
}
```

However, strcpy() can easily cause array overflows if you're not careful! In the following program, dest isn't big enough to hold the entire string, so array overflow results.

```
#include <cstring>

int main()
{
    char source[]{ "Copy this!" };
    char dest[5]; // note that the length of dest is only 5 chars!
    std::strcpy(dest, source); // overflow!
    std::cout << dest << '\n';

    return 0;
}
```

Many programmers recommend using strncpy() instead, which allows you to specify the size of the buffer, and ensures overflow doesn't occur. Unfortunately, strncpy() doesn't ensure strings are null terminated, which still leaves plenty of room for array overflow.

In C++11, strcpy_s() is preferred, which adds a new parameter to define the size of the destination. However, not all compilers support this function, and to use it, you have to define __STDC_WANT_LIB_EXT1__ with integer value 1.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <cstring> // for strcpy_s
int main()
{
    char source[]{ "Copy this!" };
    char dest[5]; // note that the length of dest is only 5 chars!
    strcpy_s(dest, 5, source); // A runtime error will occur in debug mode
    std::cout << dest << '\n';

    return 0;
}
```

Because not all compilers support strcpy_s(), strcpy() is a popular alternative -- even though it's non-standard, and thus not included in a lot of compilers. It also has its own set of issues. In short, there's no universally recommended solution here if you need to copy C-style string.

Another useful function is the strlen() function, which returns the length of the C-style string (without the null terminator).

```
#include <iostream>
#include <cstring>
#include <iterator> // for std::size

int main()
{
    char name[20]{ "Alex" }; // only use 5 characters (4 letters + null terminator)
    std::cout << "My name is: " << name << '\n';
    std::cout << name << " has " << std::strlen(name) << " letters.\n";
    std::cout << name << " has " << std::size(name) << " characters in the array.\n"; // use
 sizeof(name) / sizeof(name[0]) if not C++17 capable

    return 0;
}
```

The above example prints:

```
My name is: Alex
Alex has 4 letters.
Alex has 20 characters in the array.
```

Note the difference between strlen() and std::size(). strlen() prints the number of characters before the null terminator, whereas std::size (or the sizeof() trick) returns the size of the entire array, regardless of what's in it.

Other useful functions:
strcat() -- Appends one string to another (dangerous)
strncat() -- Appends one string to another (with buffer length check)
strcmp() -- Compare two strings (returns 0 if equal)
strncmp() -- Compare two strings up to a specific number of characters (returns 0 if equal)

Here's an example program using some of the concepts in this lesson:

```
#include <cstring>
#include <iostream>
#include <iterator> // for std::size

int main()
{
```

```
    // Ask the user to enter a string
    char buffer[255];
    std::cout << "Enter a string: ";
    std::cin.getline(buffer, std::size(buffer));

    int spacesFound{ 0 };
    int bufferLength{ static_cast<int>(std::strlen(buffer)) };
    // Loop through all of the characters the user entered
    for (int index{ 0 }; index < bufferLength; ++index)
    {
        // If the current character is a space, count it
        if (buffer[index] == ' ')
            ++spacesFound;
    }

    std::cout << "You typed " << spacesFound << " spaces!\n";

    return 0;
}
```

Note that we put `strlen(buffer)` outside the loop so that the string length is only calculated once, not every time the loop condition is checked.

Don't use C-style strings

It is important to know about C-style strings because they are used in a lot of code. However, now that we've explained how they work, we're going to recommend that you avoid them altogether whenever possible! Unless you have a specific, compelling reason to use C-style strings, use std::string (defined in the <string> header) instead. std::string is easier, safer, and more flexible. In the rare case that you do need to work with fixed buffer sizes and C-style strings (e.g. for memory-limited devices), we'd recommend using a well-tested 3rd party string library designed for the purpose, or `std::string_view`, which is covered in the next lesson, instead.

Rule

Use `std::string` or `std::string_view` (next lesson) instead of C-style strings.

 **6.6a -- An introduction to std::string_view**

 **Index**

 **6.5 -- Multidimensional Arrays**

C++ TUTORIAL | PRINT THIS POST

192 comments to 6.6 — C-style strings

« Older Comments 1 2 3

Potedeo
January 22, 2020 at 2:53 am · Reply

Because not all compilers support strcpy_s(), strlcpy() is a popular alternative.
What a nice new std function :)
I'm pretty sure you wanted to say "strcpy_s(), strcpy()"

nascardriver
January 22, 2020 at 4:57 am · Reply

Thanks potedleo ;) Lesson updated

Ged
November 17, 2019 at 10:03 am · Reply

I have a program that I need to make. It looks for a specific character in a long text and prints how many times it was used. I can check if the program is OK online on the website (it uses a C++11 / C++14 compiler). The programs works, but I get another problem "Time Limit Exceeded". It takes about a second for the program to run and I need it to be faster. Do you have any idea what could help out? Thanks in advance.

```
#include <iostream>
#include <fstream>
#include <cstring>


int main()
{
    std::ifstream fd("U1.txt");
    char word[20];
    int characterNumber = 0;

    while(!fd.eof())
    {
        fd >> word;
        int length = static_cast<int>(strlen(word));
        for(int index = 0; index < length; ++index)
        {
            if(word[index] == 'a') ++characterNumber;
        }
    }
    fd.close();

    std::ofstream fr("U1rez.txt");
    fr << "a " << characterNumber << '\n';
    fr.close();

    return 0;
}
```

nascardriver
November 18, 2019 at 3:44 am · Reply

Formatted extraction via `operator>>` and `std::strlen` are slow.
Your code is unsafe, as `word` will overflow if a word is longer than 20 characters.

`operator>>` can be replaced with `read()`, which doesn't do any formatting.
`std::strlen` is unnecessary, you know how long your buffer is.

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>

int main()
{
  std::ifstream ifs{ "U1.txt" };

  constexpr std::streamsize k_sBufferSize{ 1024 };

  std::ptrdiff_t iCharacterCount{ 0 };
  char buf[k_sBufferSize]{};

  while (ifs.read(buf, k_sBufferSize))
  {
    iCharacterCount += std::count(buf, buf + ifs.gcount(), 'a');
  }

  std::cout << iCharacterCount << '\n';

  return 0;
}
```

You can play around with the buffer size. Larger buffers should be faster. This code should be at least twice as fast as yours.

Wallace
October 18, 2019 at 7:45 am · Reply

A few minor typos:

"An" should be "A":

```
    strcpy_s(dest, 5, source); // An runtime error will occur in debug mode
```

"compiler" should be "compilers":
"Because not all compiler support strcpy_s()"

"it's" should be "its":
"It also has it’s own set of issues."

Thanks for a great resource!

Alex
October 18, 2019 at 2:51 pm · Reply

Typos fixed. Thanks!

alfonso
September 15, 2019 at 11:34 pm · Reply

"Because not all compiler support strcpy_s()..."

I do not have strcpy_s (). I thought standard libraries are the same. But it looks like someone add new functionality to a standard library and someone else do not.

nascardriver
September 16, 2019 at 11:33 pm · Reply

The C++ standard defines some functions and types that are optional. Implementations with and without those entities are both valid. To guarantee that your code works everywhere, avoid those entities altogether unless you provide alternatives.

alfonso
September 21, 2019 at 11:56 pm · Reply

To be even more confusing ...
I have Arch Linux on 64 bit, CodeBlocks (using clang or gcc).
If I use CodeBlocks, I can use std::size() (from <iterator> header) with both clang and gcc.
If I use the terminal with clang++ or g++, std::size() (from <iterator> header) is not found or something like that.
But CodeBlocks use the same clang (or gcc) that is installed on Arch Linux... 0_O

nascardriver
September 21, 2019 at 11:59 pm · Reply

You need to tell clang which standard to use

```
clang++ -std=c++17 ./main.cpp
```

alfonso
September 15, 2019 at 11:30 pm · Reply

Why is sizeof () always the same? On my system 32.

```
std::string a {"aaaaaaaaaaaaaaaaaaaaaaa"};
std::string b {"bbb"};

std::cout << sizeof (a) << '\n'; // 32
std::cout << sizeof (b) << '\n'; // 32
std::cout << sizeof (std::string) << '\n'; // 32
```

sizeof (<some_structure_identifier_here>) gives me the size of that structure in bytes, but this is not the case with std::string objects.

nascardriver
September 16, 2019 at 11:31 pm · Reply

`std::string` stores a pointer to the char array, it doesn't store the string directly. You'll understand it by the end of this chapter.

alfonso
September 21, 2019 at 1:03 am · Reply

This is somehow unexpected for a beginner, because sizeof (arr) (where arr is a fixed array) gives the whole size of the array. Even if arr decays to a pointer. Maybe because it would be almost useless to give the size of a pointer, and more valuable to give the whole size of

the array. But then sizeof (std::string) being always the same, it looks like it is also useless. Or maybe don't. So many little details in C++. :)

nascardriver
September 21, 2019 at 2:04 am · Reply

> Even if arr decays to a pointer
No. `sizeof(arr)` returns the size of the pointer if the array decayed.

`sizeof` is supposed to give you the size of the data type, and that's what it's doing. If you want to know something type-specific, eg. string length, you should use the functions provided by that type (std::string::length).
You only need `sizeof` if you're doing operations on the raw data. Use `std::size` for arrays.

alfonso
September 15, 2019 at 11:20 pm · Reply

```
for (int index = 0; index < strlen(buffer); ++index) {
    if (buffer[index] == ' ')
        spacesFound++;
}
```

Is this better? Here strlen () is not called and executed for every iteration.

```
int lenBuffer {strlen (buffer)};
for (int index = 0; index < lenBuffer; ++index) {
    if (buffer[index] == ' ')
        spacesFound++;
}
```

nascardriver
September 16, 2019 at 11:30 pm · Reply

Yes. If the condition is expensive (`std::strlen` has to loop through the string on every call) and doesn't change, you should store it in a temporary.

Benur21
August 27, 2019 at 3:00 pm · Reply

Instead of 'strcpy()', why not just 'source = dest'?

nascardriver
August 28, 2019 at 2:49 am · Reply

You'd copy the pointer, not the string. Pointers are covered in the next few lessons.

BP
July 25, 2019 at 9:44 pm · Reply

Hi,
I have a quick question. Say I was writing code to play a game of hangman.

I'm guessing for that game, I will need to compare every single letter of a word.
How would I do this using std::string?

Thanks!

nascardriver
July 26, 2019 at 9:16 am · Reply

Use a for-loop. Since you didn't post any code, this is what I suppose you're stuck at

```
// @strSolution is the word that the user is supposed to find
// @chUserGuess is the character entered by the user

for (std::string::size_type ui{ 0 }; ui < strSolution.length(); ++ui)
{
  if (strSolution[ui] == chUserGuess)
  {
    // ...
  }
}
```

I don't want to spoil you, so I'm not posting the rest of the code. If you have another question, feel free to ask. If you just want to see the start of 1 possible implementation without trying yourself, have a look here
https://godbolt.org/z/jB60I6

BP
July 26, 2019 at 9:30 am · Reply

Hi thanks for responding!
I am not really coding that game, it was just an example in my mind.
I was wondering how to look at the single letters in a string without c-style strings.
And to be honest, I don't get most of your example.

I'm guessing std::string::size_type is a special type of string?
and if strSolution is the word in std::string type then what is strSolution.length()? is it just the length of that word?
Are you jumping letter with ui?
Is this material explained later in this tutorial?

Thanks!

nascardriver
July 26, 2019 at 9:38 am · Reply

Chapter 17 is all about `std::string`. (Watch out, it's an old chapter with bad practices).

`std::string::size_type` is the type used by `std::string` for indexing. It's some unsigned integer type.

`std::string::length` (same as `std::string::size`) returns the length of the string.
"Hello" -> 5
"BP" -> 2

ui starts at 0 and goes all the way up to the string's length minus 1. For example, given the string "Hello", ui goes 0, 1, 2, 3, 4. (5 is an invalid index).

`std::string`'s individual characters can be accessed just like those of a C-style string, using `str[index]`.

BP
July 30, 2019 at 2:55 am · Reply

So since I still need to learn lot's or stuff before chapter 17, I thought I would just for the fun of it try using C-style strings for hangman.

It is a bit ugly, and spaghetti like, but here is my attempt.
I hope it is a bit ok...

```
1  #include <iostream>
2  #include <limits>
3  #include "Header.h"
4  #include <string>            //for std::string
5  #include <cstdlib>            //for exit()
6  #include <ctime>            //for std::time
7  #include <random>            //for the prng merssene
8  #include <iterator>            //for std::size()
9  #include <algorithm>        //for std::swap()
10
11 char getCharInput()
12 {
13     std::cout << "Please enter a char: ";
14     while (true)
15     {
16         char input{};
17         std::cin >> input;
18
19         if (std::cin.fail())
20         {
21             std::cout << "Please try again\n";
22             std::cin.clear();
23             std::cin.ignore(std::numeric_limits<std::streamsize>::max(
24 ), '\n');
25         }
26
27         else
28         {
29             std::cin.ignore(std::numeric_limits<std::streamsize>::max(
30 ), '\n');
31             return input;
32         }
33     }
34 }
35
36 std::string getStringInput()
37 {
38     std::cout << "Please enter a string: ";
39     std::string input{};
40     std::getline(std::cin, input);
41     return input;
42 }
43
44 std::string getUserChoice()
45 {
46     while (true)
47     {
48         std::string input{};
49         std::getline(std::cin, input);
```

```
50
51
52
53          if ((input != "letter") && (input != "word"))
54              std::cout << "Invalid string, please try again\n";
55
56
57          else
58              return input;
59      }
60  }
61
62  int main()
63  {
64      //word to be guessed, needs to be entered before compiling
65      char word[]{ "fantastic" };
66      const int wordLength{ std::size(word) - 1 };
67
68      const int amountGuesses{ 10 };
69
70      //needed to know which letters have been guessed, and to show the
71  m.
72      bool lettersShown[wordLength]{ false };
73
74      //This could be used with a for loop to show the guessed chars, bu
75  t not words.
76      //I will not be doing that because now you can just scroll up to r
77  ead the input.
78      //char guessedChars[amountGuesses]{};
79
80      for (int guesses{ amountGuesses }; guesses > 0;)
81      {
82          //Shown your word, with the revealed letters
83          std::cout << "Your word looks like this: ";
84          int lettersGuessed{};        //counter to count the amount of
85   correctly guessed letters
86          for (int index{}; index < wordLength; ++index)
87          {
88              if (lettersShown[index])
89              {
90                  std::cout << word[index];
91                  ++lettersGuessed;
92              }
93              else
94                  std::cout << "_ ";
95          }
96
97          //If you correctly guessed all letters, you win
98          if (lettersGuessed == wordLength)
99          {
100             std::cout << "Concrats, you win!\n";
101             break;
102         }
103         else
104             lettersGuessed = 0;        //counter reset.
105
106         std::cout << "\nYou have " << guesses << " guesses left!\n";
107
108         //I don't know how I would code this without this part with ev
109 erything I know right now.
        std::cout << "For your next guess please choose one of these o
ptions: \n"
```

```
            << "\tletter\tword\n";

        std::string userChoice{ getUserChoice() };

        //User guesses word, simple string comparing
        if (userChoice == "word")
        {
            std::string userGuessString{ getStringInput() };
            if (userGuessString == word)
            {
                std::cout << "concrats, you win!";
                break;
            }
            else
            {
                std::cout << "Sorry wrong choice!\n";
                --guesses;
            }
        }

        //User guesses letter, using for loop to check every letter
        else
        {
            char userGuessChar{ getCharInput() };
            bool guessedLetter{ false };            //need this to tel
l if correct or not
            for (int index{}; index < wordLength; ++index)
            {
                if (userGuessChar == word[index])
                {
                    //++guesses;
                    //I had trouble with this, so I just removed the -
-guesses from the for loop
                    //So now I just remove guesses when something was
 incorrect.

                    lettersShown[index] = true;        //need this to
 know which letters to show
                    guessedLetter = true;
                }
            }
            if (guessedLetter)
                std::cout << "Correct!\n";
            else
            {
                std::cout << "Incorrect!\n";
                --guesses;
            }

        }

    }



    //localMain();
    //menu();
    return 0;
}
```

Thanks for all your help!

nascardriver
July 30, 2019 at 3:20 am · Reply

The only place where you used a C-style string is line 63, which is only required for the fixed-size array. Other than that your code is already using `std::string`.
You should be able to understand chapter 17, I don't know why Alex decided to wait until the end.

Some suggestions:

- Limit your lines to 80 characters in length for better readability on small displays.
- Line 19+: I don't think char extraction can fail.
- Line 69: That `false` doesn't do anything. The entire array is initialized to `false` by using empty curly brackets.
- Line 98 has no effect. `lettersGuessed` isn't used after here.
- You could remove the "word"/"letter" test by always reading a string and checking its length. If it's 1, you run the letter code. If it's > 1, you run the word code.

BP
July 30, 2019 at 4:11 am · Reply

How would I check the length of a array if it's size should be fixed?
It doesn't matter what I put in the machine the size is fixed.
I guess this can be solved by using non fixed arrays?
I'll probably see those in a few chapters.
Thanks again for your pointers!

nascardriver
July 30, 2019 at 7:52 am · Reply

`std::string` is dynamic, ie. it's length is not known at compile-time, so you'd have to use a dynamically allocated array (Covered later in this chapter).

Alex
August 3, 2019 at 3:27 pm · Reply

The core lessons of this tutorial are focused on broad understanding. Originally, the lessons at the end were going to be a deeper dive into some of the more common classes of the standard library -- but I didn't get far in that approach.

SEEMA DHONDIBHAU NARAD
July 25, 2019 at 9:43 pm · Reply

#include <iostream>
int main()
{
char name[2]; // declare array large enough to hold 255 characters
std::cout << "Enter your name: ";//mangesh

```
    std::cin.getline(name, 3);
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

Stani
July 25, 2019 at 6:26 am · Reply

Thank you so much for this wonderful tutorial! I finally feel I can learn c++!
My question is the following: I wanted to change the value of one element of the string as:

```
char ss[] = "feast";
ss[0] = "b";
std::cout << ss;
```

which gives me the error: invalid conversion from 'const char*' to 'char' [-fpermissive]|

but when I switch it to:

```
ss[0] = 'b';
```

everything compiles fine. Why?

nascardriver
July 25, 2019 at 8:36 am · Reply

"b" is a char array (const char *)
'b' is a char (char)
ss[0] is a char
You can't assign a char array to a char.

Alireza
April 22, 2019 at 8:31 pm · Reply

Hello and good morning,
why does an array outputs its address when we're using array(itself, without braces), and @string which is C-style string does not ?!
Skim below to understand.

```
1   int array[] {1,2,3,4,5,6,7,8,9,10};
2   int length = sizeof(array)/sizeof(array[0]);
3
4   std::cout << "\n\narray: " << array << "\n\n";
5
6   std::cout << "size of array: " << length << "\nmembers: ";
7   for(int var=0;var<length;++var)
8       std::cout << array[var] << "  ";
9
10  std::cout << "\n\n\n";
11
12  char string[] {"C-string is an array"};
13  int strlen = sizeof(string)/sizeof(string[0]);
14
15  std::cout << "\n\narray: " << string << "\n\n";
16
17  std::cout << "size of array: " << strlen << "\nmembers: ";
18  for(int var=0;var<strlen;++var)
19      std::cout << string[var] << "  ";
```

nascardriver
April 23, 2019 at 8:25 am · Reply

Arrays decay into pointers. @std::cout<< treats char pointers as strings.

Alireza
April 24, 2019 at 3:43 am · Reply

So the difference between C-style strings and arrays is that @std::cout treats them different, ok?
And they all are arrays with different type.
Right ?!

nascardriver
April 24, 2019 at 3:44 am · Reply

right

Dimbo1911
April 11, 2019 at 11:52 pm · Reply

I have a question. How come when you overflow the array (the example with strcpy()), it still prints out the array as it should (I can print out the string in the dest same as in the source)? The length of the dest (strlen) after copying is equal to the actual length of the dest + the length of the string that was copied, but the sizeof dest still returns declared length of the string. Is this dependent on the machine and/or the compiler? (Windows 7x64 and Code::Blocks 17.12) Does the compiler allocate the array dynamically? Also, what happens with the strlen, why is it the original length + lenght of the copied string, not just the length of the copied string OR the original length? Thank you for your time

```
1   char sauce[]{ "I want to copy" };
2   char dest[2]{};
3
4   strcpy(dest, sauce);
5   std::cout << dest << "\n";
6   // returns lengths 12 14
7   std::cout << "length is " << strlen(sauce) << "\t" << strlen(dest) << "\n";
8   // returns size 2
9   std::cout << "size of dest is: " << sizeof(dest)/sizeof(dest[0]) << "\n";
10  // returns test::I want to copy::test
11  std::cout << "test::" << dest << "::test\n";
12  // this also works but I think it shouldnt
13  std::cout << dest[0] << dest[1] << dest[2];
```

nascardriver
April 12, 2019 at 1:44 am · Reply

@std::strcpy doesn't check bounds, it copies as much as you tell it to. If you tell it to write to memory you're not supposed to access, behavior is undefined.
@sizeof returns the length you declared the array with, it can't change at run-time.
@std::strlen loops through the string until it finds the 0-terminator. Again, without bounds checks, if it accesses memory after the reserved length, behavior is undefined.

Dimbo1911
April 12, 2019 at 3:08 am · Reply

Oh, ok, so if I got it right, it is a matter of pure luck if the program will output what we gave it if we write out of bounds? Like, if some other program overwrites the memory we accesed by going out of bounds, and wrote over the /0 terminator, the program will keep outputing random garbage, untill it, by chance, finds first /0 terminator?
Thanks for the answer :)

nascardriver
April 12, 2019 at 3:12 am · Reply

> some other program overwrites the memory we accesed
That can't happen, at least not on the major OSs. But other variables or code in your program could conflict with the memory you're trying to access.
@std::cout prints everything until it finds the 0-terminator, or your program crashes, because it's accessing memory which it's not allowed to access, whichever comes first.

Dimbo1911
April 12, 2019 at 3:36 am · Reply

Thanks for clearing that up

oliver
March 15, 2019 at 6:29 pm · Reply

When I try to convert std::size(array) into an integer type, vb gives me an error and says that in order to convert from 'size_t' to' int' I need to do a narrowing conversion. However in your code you did not have to do a narrowing conversion. I changed my c++ version to c++17, and also tried changing it to the latest update, but the error message still came up. Could anyone please explain this to me? Thanks

nascardriver
March 16, 2019 at 4:18 am · Reply

Use a @static_cast.
Narrowing conversions usually cause a warning, you probably set your compiler up to treat warnings as errors (This is a good thing).

```
1 int iSize{ static_cast<int>(std::size(arr)) };
```

TheBeginer
March 13, 2019 at 11:52 am · Reply

Hello, can you please help me with this code:

```
1 #include <iostream>
2 #include<fstream>
3
4 using namespace std;
5
6 int main()
7 {
8     ifstream fin;
```

```
    ofstream fout;
    fout.open("testFile");
    cout<<"Enter a string : ";
    char line[80];
    cin.getline(line,80);
    int i{0};//counter
    while(line[i]!='\0')
    {
        fout.put(line[i]);
        i++;
    }
    fout.close();
    fin.open("testFile");
    char ch;
    while(fin.eof()==0)
    {
        fin.get(ch);
        cout<<ch;
    }

    return 0;
}
```

The problem is, when I read the string from the file, it repeats a character at the end.
e.g. If I store "welcome" in the file, it prints "welcomee" on the screen although the file contains the string "welcome".

nascardriver
March 15, 2019 at 2:26 am · Reply

* Line 8, 9, 12, 22: Initialize your variables with brace initializers.
* Line 23: Booleans are false/true.
* Line 14-19: Should be a for-loop.
* Don't use "using namespace".
* Missing printed trailing line feed.
* Line 18: Use ++prefix unless you need postfix++.
* Magic number: 80. Use @std::size.

You're not writing a null-terminator to the file.

TheBeginer
March 16, 2019 at 6:12 am · Reply

Ok thanks Nascar!

Jeroen P. Broks
February 18, 2019 at 1:48 am · Reply

What I wonder now is this... Is there a way to convert a C-string into a std::string and vice versa.... Like this pseudo code:

```
char myName[10] = "Jeroen";
std::string stdStringName;
stdStringName = fromCString(myName);
```

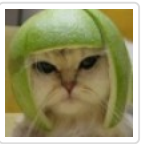
I'm not sure if I would ever actually need this, but just in case ;)

nascardriver
February 18, 2019 at 7:40 am · Reply

```
const char *szName{ "Jeroen" };
std::string strName{ szName }; // Initialize
strName = szName; // Assign
const char *szOtherName{ strName.c_str() };
// @szOtherName is not a copy of the string, it's the same!
char szCopy[7]{};
std::strcpy(szCopy, strName.c_str());
// @szCopy is a copy of the string.
```

Alamin Sheikh
February 8, 2019 at 1:40 am · Reply

std::
why its important ??



Alex
February 9, 2019 at 8:54 am · Reply

https://www.learncpp.com/cpp-tutorial/naming-conflicts-and-the-std-namespace/

Piyush
January 12, 2019 at 12:29 am · Reply

```
std::cin.getline(var , len);
std::getline(std::cin , var);
```

The first line only supports C style string variable as its first argument.
The second line only supports string variable as its second argument.
Are the getline functions same in both lines.
I am confused with it.

nascardriver
January 12, 2019 at 4:30 am · Reply

Hi Piyush!

They're not the same, but they behave the same, use whichever you prefer, the second is easier to use since you don't need to know the length beforehand.

Piyush
January 12, 2019 at 6:33 am · Reply

Thanks for the reply

« Older Comments 1 2 3