Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required. Take the 2-minute tour ×

# string c_str() vs. data()

I have read several places that the difference between `c_str()` and `data()` (in STL and other implementations) is that `c_str()` is always null terminated while `data()` is not. As far as I have seen in actual implementations, they either do the same or `data()` calls `c_str()`.

What am I missing here? Which one is more correct to use in which scenarios?

c++ stl c-str

edited Jun 2 '14 at 11:31
Wolf
1,936 ● 5 ● 27

asked Oct 11 '08 at 20:59
leon

## 5 Answers

The documentation is correct. Use `c_str()` if you want a null terminated string.

If the implementers happend to implement `data()` in terms of `c_str()` you don't have to worry, still use `data()` if you don't need the string to be null terminated, in some implementation it may turn out to perform better than c_str().

strings don't necessarily have to be composed of character data, they could be composed with elements of any type. In those cases `data()` is more meaningful. `c_str()` in my opinion is only really useful when the elements of your string are character based.

*Extra*: In C++11 onwards, both functions are required to be the same. i.e. `data` is now required to be null-terminated. According to cppreference: "The returned array is null-terminated, that is, data() and c_str() perform the same function."

edited Oct 29 '13 at 11:10
Aaron McDaid
9,901 ● 2 ● 24 ● 42

answered Oct 11 '08 at 21:01
Scott Langham
21.3k ● 20 ● 76 ● 134

{ USE STACK OVERFLOW TO FIND THE BEST DEVELOPERS } stackoverflowcareers

Even know you have seen that they do the same, or that .data() calls .c_str(), it is not correct to assume that this will be the case for other compilers. It is also possible that your compiler will change with a future release.

**2 reasons to use std::string:**

std::string can be used for both text and arbitrary binary data.

```
//Example 1
[illegible]
[illegible]
s1 = "abc";

//Example 2
//Arbitrary binary data:
std::string s2;
s2.append("a\0b\0b\0", 6);
```

THIS PAGE ISSAFE VAULT ISACCESSSITE ISINFOBAR IGNOREDOPENCLOSEDVAULTHIDDEN

You should use the .c_str() method when you are using your string as example 1.

You should use the .data() method when you are using your string as example 2. Not because it is dangereous to use .c_str() in these cases, but because it is more explicit that you are working with binary data for others reviewing your code.

**Possible pitfall with using .data()**

The following code is wrong and could cause a segfault in your program:

```
std::string s;
s = "abc";
char sz[512];
strcpy(sz, s.data());//This could crash depending on the implementation of .data()
```

**Why is it common for implementers to make .data() and .c_str() do the same thing?**

Because it is more efficient to do so. The only way to make .data() return something that is not null terminated, would be to have .c_str() or .data() copy their internal buffer, or to just use 2 buffers. Having a single null terminated buffer always means that you can always use just one internal buffer when implementing std::string.

edited Oct 11 '08 at 21:22

answered Oct 11 '08 at 21:12
Brian R. Bondy
160k ● 68 ● 411 ● 534

5 Actually, the point of .data() is that it should not copy the internal buffer. This means an implementation does not have to waste a char on \0 until it's needed. You'd never want two buffers: if you DO call .c_str(), append a \0 to the buffer. .data() can still return that buffer. – MSalters Oct 13 '08 at 10:46

1 Agreed fully it would be ridiculous to use 2 buffers. How do you know that's why .data was intended though? – Brian R. Bondy Oct 13 '08 at 14:18

---

In C++11/C++0x, `data()` and `c_str()` is no longer different. And thus `data()` is required to have a null termination at the end as well.

> 21.4.7.1 `basic_string` accessors [string.accessors]
>
> `const charT* c_str() const noexcept;`
>
> `const charT* data() const noexcept;`
>
> 1 Returns: A pointer p such that `p + i == &operator[](i)` for each `i` in `[0,size()]`.

---

> 21.4.5 basic_string element access [string.access]
>
> `const_reference operator[](size_type pos) const noexcept;`
>
> 1 Requires: pos <= size(). 2 Returns: `*(begin() + pos) if pos < size()`, otherwise a reference to an object of type T with value `charT();` the referenced value shall not be modified.

edited Oct 17 '13 at 22:46
Jamin Grey
2,940 ● 15 ● 29

answered Sep 13 '12 at 4:41
mfazekas
3,203 ● 1 ● 13 ● 17

What if the string is composed of non-character data, which is legal for string data AFAIK, including null? – taz Jul 30 '13 at 16:51

1 @taz Even when storing binary data, C++11 requires that `std::string` allocate an extra `char` for a trailing `'\0'`. When you do `std::string s("\0");`, both `s.data()[0]` and `s.data()[1]` are guaranteed to evaluate to 0. – bcrist Sep 15 '13 at 7:30

---

It has been answered already, some notes on the purpose: Freedom of implementation.

`std::string` operations - e.g. iteration, concatenation and element mutation - don't need the zero terminator. Unless you pass the `string` to a function expecting a zero terminated string, it can be omitted.

This would allow an implementation to have substrings share the actual string data: `string::substr` could internally hold a reference to shared string data, and the start/end range, avoiding the copy (and additional allocation) of the actual string data. The implementation would defer the copy until you call c_str or modify any of the strings. No copy would ever be made if the strigns involved are just read.

(copy-on-write implementation aren't much fun in multithreaded environments, plus the typical memory/allocation savings aren't worth the more complex code today, so it's rarely done).

---

Similarly, `string::data` allows a different internal representation, e.g. a rope (linked list of string segments). This can improve insert / replace operations significantly. again, the list of segments would have to be collapsed to a single segment when you call `c_str` or `data`.

answered Jul 1 '12 at 8:04
peterchen
23.5k ● 13 ● 56 ● 131

---

Quote from `ANSI ISO IEC 14882 2003` (C++03 Standard):

```
    21.3.6 basic_string string operations [lib.string.ops]

    const charT* c_str() const;

    Returns: A pointer to the initial element of an array of length size() + 1 whose first
size() elements
equal the corresponding elements of the string controlled by *this and whose last element
is a
null character specified by charT().
    Requires: The program shall not alter any of the values stored in the array. Nor shall
the program treat the
returned value as a valid pointer value after any subsequent call to a non-const member
function of the
class basic_string that designates the same object as this.

    const charT* data() const;

    Returns: If size() is nonzero, the member returns a pointer to the initial element of
an array whose first
size() elements equal the corresponding elements of the string controlled by *this. If
size() is
zero, the member returns a non-null pointer that is copyable and can have zero added to
it.
    Requires: The program shall not alter any of the values stored in the character array.
Nor shall the program
treat the returned value as a valid pointer value after any subsequent call to a non-
const member
function of basic_string that designates the same object as this.
```

answered Oct 5 '11 at 13:04
Mihran Hovsepyan
5,038 ● 3 ● 24 ● 73