

Case-insensitive string comparison in C++ [closed]

Asked 16 years, 4 months ago Modified 1 year, 3 months ago

Viewed 661k times



372



Closed. This question is [opinion-based](#). It is not currently accepting answers.

Closed 6 years ago.



This question's answers are a [community effort](#). Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

What is the best way of doing case-insensitive string comparison in C++ without transforming a string to all uppercase or all lowercase?

Please indicate whether the methods are Unicode-friendly and how portable they are.

c++

string

Share Follow

edited Aug 23, 2017 at 16:35



Toby Speight

30.6k ● 49 ● 72 ● 113

asked Aug 14, 2008 at 20:01



Adam

26.5k ● 23 ● 77 ● 87

@[Adam](#11679): While this variant is good in terms of usability it's bad in terms of performance because it creates unnecessary copies. I might overlook something but I believe the best (non-Unicode) way is to use `std::stricmp`. Otherwise, read what Herb [has to say](#). – Konrad Rudolph
Aug 26, 2008 at 12:17

In c, one usually was forced to upper the whole string then compare that way - or roll your own compare :P
– Michael Dorgan May 22, 2010 at 1:10

a later question has a simpler answer: `strcasecmp` (at least for BSD & POSIX compilers)
stackoverflow.com/questions/9182912/... – Móz Nov 5, 2013 at 21:39

@Móz this question also has that answer, with the important caveat that `strcasecmp` is not part of the standard and is missing from at least one common compiler. – Mark Ransom
Dec 1, 2014 at 19:57

Comments disabled on deleted / locked posts / reviews

30 Answers

Sorted by:

Highest score (default)





333



Boost includes a handy algorithm for this:

```
#include <boost/algorithm/string.hpp>
// Or, for fewer header dependencies:
// #include <boost/algorithm/string/predicate.hpp>

std::string str1 = "hello, world!";
std::string str2 = "HELLO, WORLD!";

if (boost::iequals(str1, str2))
{
    // Strings are identical
}
```

Share Improve this answer

Follow

edited Jun 22, 2012 at 18:28



Josh Kelley

58.3k ● 20 ● 162 ● 256

answered Nov 24, 2008 at 21:03



Rob

78.5k ● 57 ● 161 ● 199

16 Is this UTF-8 friendly? I think not. – [vladr](#) Oct 30, 2010 at 0:23

20 No, because UTF-8 allows identical strings to be coded with different binary codes, due to accents, combines, bidi issues, etc. – [vy32](#) Jun 18, 2011 at 23:35

12 @vy32 That is absolutely incorrect! The UTF-8 combinations are mutually exclusive. It must always use shortest possible representation, if it does not, it's a malformed UTF-8 sequence or code point that must be treated with care. – [Wiz](#) Nov 10, 2011 at 23:44 ✎

57 @Wiz, you are ignoring the issue of Unicode string normalization. ñ can be represented as a combining ~

followed by an n, or with a ñ character. You need to use Unicode string normalization before performing the comparison. Please review Unicode Technical Report #15, unicode.org/reports/tr15 – vy32 Nov 11, 2011 at 3:21

14 @wonkorealtime: because "ß" converted to uppercase is "SS": fileformat.info/info/unicode/char/df/index.htm
– Mooing Duck May 29, 2014 at 23:11 ✎



The trouble with boost is that you have to link with and depend on boost. Not easy in some cases (e.g. android).

150



And using char_traits means *all* your comparisons are case insensitive, which isn't usually what you want.



This should suffice. It should be reasonably efficient. Doesn't handle unicode or anything though.



```
#include <cctype>      // std::tolower
#include <algorithm>    // std::equal

bool ichar_equals(char a, char b)
{
    return std::tolower(static_cast<unsigned char>(a))
           std::tolower(static_cast<unsigned char>(b))
}

bool iequals(const std::string& a, const std::string&
{
    return a.size() == b.size() &&
           std::equal(a.begin(), a.end(), b.begin(), i
}
```

C++14 version

```
#include <cctype>      // std::tolower
#include <algorithm>    // std::equal

bool iequals(const std::string& a, const std::string&
{
    return std::equal(a.begin(), a.end(), b.begin(), b
}
```

C++20 version using `std::ranges`

```
#include <cctype>      // std::tolower
#include <algorithm>    // std::equal
#include <string_view>  // std::string_view

bool iequals(std::string_view lhs, std::string_view rhs
{
    return std::ranges::equal(lhs, rhs, icar_equals);
}
```

Share Improve this answer

edited Sep 13, 2023 at 9:54

Follow



Jan Schultke

38.3k ● 8 ● 87 ● 168

answered Nov 7, 2010 at 21:49



Timmmm

96k ● 78 ● 401 ● 568

33 Actually, the boost string library is a header only library, so there is no need to link to anything. Also, you can use boost's 'bcp' utility to copy just the string headers to your source tree, so you don't need to require the full boost library.

– [Gretchen](#) Mar 9, 2011 at 21:47

2 @Anna Text library of boost needs to be built and link. It uses IBM ICU. – [Behrouz.M](#) Jun 1, 2015 at 6:46

6 `std::tolower` should not be called on `char` directly, a `static_cast` to `unsigned char` is needed. – [Evg](#) Sep 26, 2020 at 9:50 ✎

1 In the C++14 version, it suffices to change to parameter list of the lambda function to `[](unsigned char a, unsigned char b)`, no `static_cast` is necessary. – [Jonatan Lindén](#) Sep 16, 2021 at 11:08

2 @Timmeee I've taken the liberty of adding a C++20 version to this answer as I believe here is the best fit, and compared with other answers in this thread, I feel most closely resembles your other solutions. – [Ben Cottrell](#) Jan 2, 2022 at 11:39 ✎



123



Take advantage of the standard `char_traits`. Recall that a `std::string` is in fact a typedef for

`std::basic_string<char>`, or more explicitly, `std::basic_string<char, std::char_traits<char> >`.

The `char_traits` type describes how characters compare, how they copy, how they cast etc. All you need to do is typedef a new string over `basic_string`, and provide it with your own custom `char_traits` that compare case insensitively.

```
struct ci_char_traits : public char_traits<char> {
    static bool eq(char c1, char c2) { return toupper(
    static bool ne(char c1, char c2) { return toupper(
    static bool lt(char c1, char c2) { return toupper(
    static int compare(const char* s1, const char* s2,
        while( n-- != 0 ) {
            if( toupper(*s1) < toupper(*s2) ) return -
            if( toupper(*s1) > toupper(*s2) ) return 1
            ++s1; ++s2;
        }
}
```

```

        return 0;
    }
    static const char* find(const char* s, int n, char
        while( n-- > 0 && toupper(*s) != toupper(a) )
            ++s;
        }
        return s;
    }
};

typedef std::basic_string<char, ci_char_traits> ci_str

```

The details are on [Guru of The Week number 29](#).

Share Improve this answer

edited May 22, 2010 at 2:16

Follow

answered May 22, 2010 at 1:36



wilhelmtell

58.6k ● 20 ● 97 ● 131

15 As far as I know from my own experimentation, this makes your new string type incompatible with `std::string`.
– [Zan Lynx](#) Sep 26, 2012 at 21:25

11 Of course it does - for its own good. A case-insensitive string is something else: `typedef std::basic_string<char, ci_char_traits<char> > istring`, not `typedef std::basic_string<char, std::char_traits<char> > string`.
– [Andreas Spindler](#) Oct 9, 2012 at 9:24

301 "All you need to do..." – [Tim MB](#) Apr 19, 2013 at 10:03

36 Any language construct that forces such insanity in this trivial case should and can be abandoned without regrets.
– [Erik Aronesty](#) Nov 14, 2014 at 14:17

5 @DaveKennedy I think Erik advises abandoning human languages, as *those* are the language constructs that are forcing this insanity. :-) – [srm](#) Mar 21, 2018 at 16:35



67



If you are on a POSIX system, you can use [strcasecmp](#). This function is not part of standard C, though, nor is it available on Windows. This will perform a case-insensitive comparison on 8-bit chars, so long as the locale is POSIX. If the locale is not POSIX, the results are undefined (so it might do a localized compare, or it might not). A wide-character equivalent is not available.

Failing that, a large number of historic C library implementations have the functions `stricmp()` and `strnicmp()`. Visual C++ on Windows renamed all of these by prefixing them with an underscore because they aren't part of the ANSI standard, so on that system they're called [_stricmp](#) or [_strnicmp](#). Some libraries may also have wide-character or multibyte equivalent functions (typically named e.g. `wcsicmp`, `mbcsicmp` and so on).

C and C++ are both largely ignorant of internationalization issues, so there's no good solution to this problem, except to use a third-party library. Check out [IBM ICU \(International Components for Unicode\)](#) if you need a robust library for C/C++. ICU is for both Windows and Unix systems.

Share Improve this answer

edited Jun 4, 2015 at 14:13

Follow



al45tair

4,433 ● 24 ● 31

answered Aug 14, 2008 at 20:46



Derek Park

46.8k ● 16 ● 59 ● 76



58



Are you talking about a dumb case insensitive compare or a full normalized Unicode compare?

A dumb compare will not find strings that might be the same but are not binary equal.

Example:

U212B (ANGSTROM SIGN)
U0041 (LATIN CAPITAL LETTER A) + U030A (COMBINING RING)
U00C5 (LATIN CAPITAL LETTER A WITH RING ABOVE).

Are all equivalent but they also have different binary representations.

That said, [Unicode Normalization](#) should be a mandatory read especially if you plan on supporting Hangul, Thai and other asian languages.

Also, IBM pretty much patented most optimized Unicode algorithms and made them publicly available. They also maintain an implementation : [IBM ICU](#)

Share Improve this answer

edited Apr 13, 2020 at 11:53

Follow



Community Bot

1 ● 1

answered Aug 14, 2008 at 20:31



Coincoin

28.5k ● 7 ● 56 ● 76

Late commen, I know... *'Are all equivalent'* might not be fully correct, though I'm not familiar with the given case – German 'Umlaut's, though, could be created by combining `a`, `o` or `u` with diaeresis or directly via letters `ä`, `ö`, `ü` – *however* the distance of the two dots is (slightly) different (direct charachters narrower)... – [Aconcagua](#) Jul 6, 2022 at 16:08



`boost::iequals` is not utf-8 compatible in the case of string. You can use [boost::locale](#).

35



```
comparator<char, collator_base::secondary> cmpr;  
cout << (cmpr(str1, str2) ? "str1 < str2" : "str1 >= s
```



- Primary -- ignore accents and character case, comparing base letters only. For example "facade" and "Façade" are the same.
- Secondary -- ignore character case but consider accents. "facade" and "façade" are different but "Façade" and "façade" are the same.
- Tertiary -- consider both case and accents: "Façade" and "façade" are different. Ignore punctuation.
- Quaternary -- consider all case, accents, and punctuation. The words must be identical in terms of Unicode representation.

- Identical -- as quaternary, but compare code points as well.

Share Improve this answer

answered Apr 26, 2012 at 8:51

Follow



Igor Milyakov

612 ● 5 ● 7



My first thought for a non-unicode version was to do something like this:

35



```
bool caseInsensitiveStringCompare(const string& str1,
    if (str1.size() != str2.size()) {
        return false;
    }
    for (string::const_iterator c1 = str1.begin(), c2
str1.end(); ++c1, ++c2) {
        if (tolower(static_cast<unsigned char>(*c1)) !
tolower(static_cast<unsigned char>(*c2))) {
            return false;
        }
    }
    return true;
}
```

Share Improve this answer

edited Sep 27, 2020 at 13:27

Follow

answered Aug 26, 2008 at 11:51



Shadow2531

12.2k ● 5 ● 38 ● 51



You can use `strcasecmp` on Unix, or `stricmp` on Windows.

29



One thing that hasn't been mentioned so far is that if you are using `std::string` with these methods, it's useful to first compare the length of the two strings, since this information is already available to you in the `string` class. This could prevent doing the costly string comparison if the two strings you are comparing aren't even the same length in the first place.

Share Improve this answer

Follow

edited Jul 26, 2013 at 15:17



Yu Hao

122k ● 48 ● 246 ● 302

answered Dec 2, 2008 at 0:51



bradtgmurray

14.3k ● 10 ● 39 ● 36

Since determining the length of a string consists of iterating over every character in the string and comparing it against 0, is there really that much difference between that and just comparing the strings right away? I guess you get better memory locality in the case where both strings don't match, but probably nearly 2x runtime in case of a match.


– [uliwitness](#) Jan 22, 2014 at 13:28

6 C++11 specifies that the complexity of `std::string::length` must be constant: cplusplus.com/reference/string/string/length

– [bradtgmurray](#) Feb 4, 2014 at 21:37

2 That's a fun little fact, but has little bearing here. `strcasecmp()` and `stricmp()` both take undecorated C strings,

so there is no `std::string` involved. – [uliwitness](#) Feb 5, 2014 at 17:39

- 5 These methods will return -1 if you compare "a" vs "ab". The lengths are different but "a" comes before "ab". So, simply comparing the lengths is not feasible if the caller cares about ordering. – [Nathan](#) Feb 12, 2014 at 23:33 
-



I'm trying to cobble together a good answer from all the posts, so help me edit this:

17



Here is a method of doing this, although it does transforming the strings, and is not Unicode friendly, it should be portable which is a plus:



```
bool caseInsensitiveStringCompare( const std::string&
str2 ) {
    std::string str1Cpy( str1 );
    std::string str2Cpy( str2 );
    std::transform( str1Cpy.begin(), str1Cpy.end(), str2Cpy.begin(), str2Cpy.end(),
);
    std::transform( str2Cpy.begin(), str2Cpy.end(), str1Cpy.begin(), str1Cpy.end(),
);
    return ( str1Cpy == str2Cpy );
}
```

From what I have read this is more portable than `stricmp()` because `stricmp()` is not in fact part of the `std` library, but only implemented by most compiler vendors.

To get a truly Unicode friendly implementation it appears you must go outside the `std` library. One good 3rd party library is the [IBM ICU \(International Components for Unicode\)](#).

Also `boost::iequals` provides a fairly good utility for doing this sort of comparison.

Share Improve this answer

edited Dec 19, 2011 at 21:54

Follow

community wiki

7 revs

Adam

can you please tell, what does mean `::tolower`, why you can use `tolower` instead of `tolower()`, and what is `::` before?

thanks – [VextoR](#) Mar 11, 2011 at 8:40 

22 This is not a very efficient solution - you make copies of both strings and transform all of them even if the first character is different. – [Timm](#) Mar 13, 2011 at 18:14

3 If you're going to make a copy anyway, why not pass by value instead of by reference? – [celticminstrel](#) Jun 21, 2015 at 2:17

1 the question asks explicitly to not `transform` the whole string before comparison – [Sandburg](#) Jun 6, 2019 at 15:43

1 `std::tolower` should not be called on `char` directly, a `static_cast` to `unsigned char` is needed. – [Evg](#) Sep 26, 2020 at 9:53



```
str1.size() == str2.size() && std::equal(str1.begin(),
str2.begin(), [](auto a, auto b){return std::tolower(a
```



You can use the above code in C++14 if you are not in a position to use boost. You have to use `std::tolower` for wide chars.



Share Improve this answer

edited Oct 21, 2019 at 15:06

Follow



bcmpinc

3,350 ● 30 ● 38

answered Apr 5, 2017 at 9:18



vine'th

5,010 ● 2 ● 30 ● 28

4 I think you need to add a `str1.size() == str2.size()` && to the front so that will not go out of bounds when str2 is a prefix of str1. – [neuroburn](#) Aug 1, 2017 at 14:06



See [std::lexicographical_compare](#):

16



```
// lexicographical_compare example
#include <iostream>      // std::cout, std::boolalpha
#include <algorithm>      // std::lexicographical_compar
#include <cctype>         // std::tolower

// a case-insensitive comparison function:
bool mycomp(char c1, char c2) {
    return std::tolower(c1) < std::tolower(c2);
}

int main() {
    std::string foo = "Apple";
    std::string bar = "apartment";

    std::cout << std::boolalpha;

    std::cout << "Comparing foo and bar lexicographica
```

```

    std::cout << "Using default comparison (operator<)\n";
    std::cout << std::lexicographical_compare(foo.begin(),
bar.begin(), bar.end());
    std::cout << '\n';

    std::cout << "Using custom comparison (mycomp): ";
    std::cout << std::lexicographical_compare(foo.begin(),
bar.begin(), bar.end(), mycomp);
    std::cout << '\n';

    return 0;
}

```

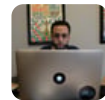
[Demo](#)

Share Improve this answer

edited Jun 6, 2023 at 17:38

Follow

answered Sep 16, 2015 at 21:54



Brian Rodriguez

4,349 ● 1 ● 18 ● 41


1 This method is potentially unsafe and non-portable. `std::tolower` works only if the character is ASCII-encoded. There is no such guarantee for `std::string` - so it can be undefined behavior easily. – [plasmacel](#) Mar 27, 2018 at 14:27

1 @plasmacel Then use a function that works w/ other encodings. – [Brian Rodriguez](#) Apr 6, 2018 at 15:05

`std::lexicographical_compare` looked so promising, until we got to `mycomp`. :-(– [Robin Davies](#) Mar 12, 2023 at 7:52

This algorithm works on any element type. If you want unicode support, then provide unicode-aware strings with unicode-aware iterators/comparisons. In other words, the

issue lies with `std::string` not with

`std::lexicographical_compare`. – Brian Rodriguez Jun 6, 2023 at 17:28 



Short and nice. No other dependencies, than *extended* std C lib.

14



```
strcasecmp(str1.c_str(), str2.c_str()) == 0
```



returns **true** if `str1` and `str2` are equal. `strcasecmp` may not exist, there could be analogs `stricmp`, `strcmpi`, etc.

Example code:

```
#include <iostream>
#include <string>
#include <string.h> //For strcasecmp(). Also could be

using namespace std;

/// Simple wrapper
inline bool str_ignoreCase_cmp(std::string const& s1,
    if(s1.length() != s2.length())
        return false; // optimization since std::stri
variable.
    return strcasecmp(s1.c_str(), s2.c_str()) == 0;
}

/// Function object - comparator
struct StringCaseInsensitiveCompare {
    bool operator()(std::string const& s1, std::string
        if(s1.length() != s2.length())
            return false; // optimization since std::
variable.
        return strcasecmp(s1.c_str(), s2.c_str()) == 0
```

```

    }
    bool operator()(const char *s1, const char * s2){
        return strcasecmp(s1,s2)==0;
    }
};

/// Convert bool to string
inline char const* bool2str(bool b){ return b?"true":""

int main()
{
    cout<< bool2str(strcasecmp("asd", "AsD")==0) <<endl
    cout<<
    bool2str(strcasecmp(string{"aasd"}.c_str(),string{"Aas
    StringCaseInsensitiveCompare cmp;
    cout<< bool2str(cmp("A","a")) <<endl;
    cout<< bool2str(cmp(string{"Aaaa"},string{"aaaA"}))
    cout<< bool2str(str_ignoreCase_cmp(string{"Aaaa"},
    return 0;
}

```

Output:

```

true
true
true
true
true

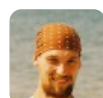
```

Share Improve this answer

edited Oct 21, 2016 at 19:05

Follow

answered Sep 30, 2016 at 15:51





kyb

8,071 ● 10 ● 63 ● 116

11 it is strange that C++ `std::string` has no ignore-case comparison method.. – [kyb](#) Sep 30, 2016 at 15:52

1 "strcasecmp is not part of the standard" - Mark Ransom Dec 1 '14 at 19:57 – [Liviu](#) Oct 21, 2016 at 14:21

1 yes, but the most of modern compilers have it or its another-named analog. `stricmp`, `strcmpi`, `strcasecmp`, etc. Thank you. message edited. – [kyb](#) Oct 21, 2016 at 19:01 

TODO: use `cout << boolalpha` rather than my `bool2str` because It to implicitly convert bool to chars for stream. – [kyb](#) Jun 1, 2017 at 12:15 

It's in `<strings.h>` in gcc's libraries. – [Owl](#) Aug 7, 2017 at 21:09



Visual C++ string functions supporting unicode:

<http://msdn.microsoft.com/en-us/library/cc194799.aspx>

13

the one you are probably looking for is `_wcsnicmp`



Share Improve this answer

edited Jul 26, 2013 at 15:17



Follow



Yu Hao

122k ● 48 ● 246 ● 302



answered Aug 14, 2008 at 20:11



Darren Kopp

77.6k ● 9 ● 80 ● 93

7 Ironically, Microsoft's "wide character codes" are NOT unicode clean because they do not handle unicode normalization. – [vy32](#) Jun 18, 2011 at 23:36



FYI, `strcmp()` and `stricmp()` are vulnerable to buffer overflow, since they just process until they hit a null terminator. It's safer to use `_strncmp()` and

12

[`_strnicmp\(\)`](#).



Share Improve this answer

edited Jul 26, 2013 at 15:18



Follow



Yu Hao

122k ● 48 ● 246 ● 302



answered Aug 14, 2008 at 20:29



Wedge

19.8k ● 7 ● 50 ● 71

6 True, although overREADING a buffer is significantly less dangerous than overWRITEing a buffer. – [Adam Rosenfield](#) Nov 17, 2008 at 20:47

4 `stricmp()` and `strnicmp()` are not part of the POSIX standard :-(However you can find `strcasecmp()`, `strcasecmp_l()`, `strncasecmp()` and `strncasecmp_l()` in POSIX header `strings.h` :-) see opengroup.org – [oHo](#) Apr 11, 2013 at 12:27 ✎

4 @AdamRosenfield 'worse' depends on context. In security, sometimes the whole point of an overwrite is to get to overread. – [karmakaze](#) Mar 21, 2015 at 16:22



The [Boost.String](#) library has a lot of algorithms for doing case-insensitive comparisons and so on.

11



You could implement your own, but why bother when it's already been done?



Share Improve this answer

answered May 22, 2010 at 0:57



Follow



[Dean Harding](#)

72.6k ● 13 ● 148 ● 180

1 There isn't a way built-in with `std::string`? – [WilliamKF](#) May 22, 2010 at 0:58

6 No, there isn't. – [Dean Harding](#) May 22, 2010 at 1:06

3 "... why bother when it's already been done?" - what if you are not using Boost? The OP did not have the tag with the question. – [jww](#) Feb 7, 2016 at 22:07



9



For my basic case insensitive string comparison needs I prefer not to have to use an external library, nor do I want a separate string class with case insensitive traits that is incompatible with all my other strings.

So what I've come up with is this:

```
bool icasecmp(const string& l, const string& r)
{
    return l.size() == r.size()
        && equal(l.cbegin(), l.cend(), r.cbegin(),
            [](string::value_type l1, string::value_ty
                { return toupper(l1) == toupper(r1); }
        )
}

bool icasecmp(const wstring& l, const wstring& r)
{
    return l.size() == r.size()
        && equal(l.cbegin(), l.cend(), r.cbegin(),
            [](wstring::value_type l1, wstring::value_
                { return toupper(l1) == toupper(r1); }
        )
}
```

A simple function with one overload for char and another for wchar_t. Doesn't use anything non-standard so should be fine on any platform.

The equality comparison won't consider issues like variable length encoding and Unicode normalization, but basic_string has no support for that that I'm aware of anyway and it isn't normally an issue.

In cases where more sophisticated lexicographical manipulation of text is required, then you simply have to

use a third party library like Boost, which is to be expected.

Share Improve this answer

edited Jun 28, 2013 at 15:46

Follow

answered Jun 26, 2013 at 21:29



Neutrino

9,542 ● 9 ● 65 ● 99

2 You could probably make that one function if you made it a template and used `basic_string<T>` instead of separate `string/wstring` versions? – [uliwitness](#) Jan 22, 2014 at 13:31

2 How would the single function template invoke either `toupper` or `tolower` without resorting to use of specialization or macros, a function overload seems like a simpler and more appropriate implementation than either. – [Neutrino](#) Jun 28, 2015 at 15:39



9



Doing this without using Boost can be done by getting the C string pointer with `c_str()` and using `strcasecmp`:

```
std::string str1 = "aBcD";
std::string str2 = "AbCd";
if (strcasecmp(str1.c_str(), str2.c_str()) == 0)
{
    //case insensitive equal
}
```

Share Improve this answer

edited Jan 12, 2016 at 9:19

Follow

answered Jan 12, 2016 at 9:08



DavidS

2,284 ● 1 ● 21 ● 24



Assuming you are looking for a method and not a magic function that already exists, there is frankly no better way.

6



We could all write code snippets with clever tricks for limited character sets, but at the end of the day at somepoint you have to convert the characters.



The best approach for this conversion is to do so prior to the comparison. This allows you a good deal of flexibility when it comes to encoding schemes, which your actual comparison operator should be ignorant of.

You can of course 'hide' this conversion behind your own string function or class, but you still need to convert the strings prior to comparison.

Share Improve this answer

edited Aug 14, 2008 at 20:13

Follow

answered Aug 14, 2008 at 20:10



Andrew Grant

58.8k ● 22 ● 131 ● 144



I wrote a case-insensitive version of `char_traits` for use with `std::basic_string` in order to generate a `std::string` that is not case-sensitive when doing comparisons,

6



searches, etc using the built-in `std::basic_string` member functions.



So in other words, I wanted to do something like this.



```
std::string a = "Hello, World!";  
std::string b = "hello, world!";  
  
assert( a == b );
```

...which `std::string` can't handle. Here's the usage of my new `char_traits`:

```
std::istring a = "Hello, World!";  
std::istring b = "hello, world!";  
  
assert( a == b );
```

...and here's the implementation:

```
/* ---  
  
    Case-Insensitive char_traits for std::string's  
  
    Use:  
  
        To declare a std::string which preserves c  
comparisons & search,  
        use the following syntax:  
  
        std::basic_string<char, char_traits_no  
noCaseString;  
  
        A typedef is declared below which simplifi  
  
        typedef std::basic_string<char, char_t  
istring;
```

```
--- */
```

```
template<class C>
struct char_traits_nocase : public std::char_traits<C>
{
    static bool eq( const C& c1, const C& c2 )
    {
        return ::toupper(c1) == ::toupper(c2);
    }

    static bool lt( const C& c1, const C& c2 )
    {
        return ::toupper(c1) < ::toupper(c2);
    }

    static int compare( const C* s1, const C* s2,
                        rsize_t N)
    {
        return _strnicmp(s1, s2, N);
    }

    static const char* find( const C* s, rsize_t N,
                             const C& a)
    {
        for( rsize_t i=0 ; i<N ; ++i )
        {
            if( ::toupper(s[i]) == ::toupper(a) )
                return s+i ;
        }
        return 0 ;
    }

    static bool eq_int_type( const int_type& c1, const int_type& c2 )
    {
        return ::toupper(c1) == ::toupper(c2) ;
    }
};
```

```
template<>
struct char_traits_nocase<wchar_t> : public std::char_traits<wchar_t>
{
    static bool eq( const wchar_t& c1, const wchar_t& c2 )
    {
        return ::towupper(c1) == ::towupper(c2);
    }
};
```

```

static bool lt( const wchar_t& c1, const wchar
{
    return ::towupper(c1) < ::towupper(c2);
}

static int compare( const wchar_t* s1, const w
{
    return _wcsnicmp(s1, s2, N);
}

static const wchar_t* find( const wchar_t* s,
a )
{
    for( size_t i=0 ; i<N ; ++i )
    {
        if( ::towupper(s[i]) == ::towupper(a)
            return s+i ;
        }
    }
    return 0 ;
}

static bool eq_int_type( const int_type& c1, c
{
    return ::towupper(c1) == ::towupper(c2) ;
}
};

typedef std::basic_string<char, char_traits_nocase
typedef std::basic_string<wchar_t, char_traits_noc

```

Share Improve this answer

answered Nov 17, 2008 at 23:32

Follow



John Dibling

101k ● 32 ● 191 ● 331

-
- 2 This works for regular chars, but won't work for all of Unicode, as capitalization is not necessarily bidirectional (there's a good example in Greek involving sigma that I can't remember right now; something like it has two lower and one

upper case, and you can't get a proper comparison either way) – [coppro](#) Nov 24, 2008 at 21:02

- 1 That's really the wrong way to go about it. Case sensitivity should not be a property of the strings themselves. What happens when the same string object needs both case-sensitive and case insensitive comparisons? – [Ferruccio](#) Nov 24, 2008 at 21:07
-

If case-sensitivity isn't appropriate to be "part of" the string, then neither is the `find()` function at all. Which, for you, might be true, and that's fine. IMO the greatest thing about C++ is that it doesn't force a particular paradigm on the programmer. It is what you want/need it to be. – [John Dibling](#) Nov 25, 2008 at 6:20

Actually, I think most C++-guru's (like the ones on the standards committee) agree that it was a mistake to put `find()` in `std::basic_string<>` along with a whole lot of other things that could equally well be placed in free functions. Besides there are some issues with putting it in the type.
– [Andreas Magnusson](#) Nov 25, 2008 at 7:50

As others have pointed out, there are two major things wrong with this solution (ironically, one is the interface and the other is the implementation ;-)). – [Konrad Rudolph](#) Nov 25, 2008 at 8:15



Late to the party, but here is a variant that uses `std::locale`, and thus correctly handles Turkish:

6



```
auto tolower = std::bind1st(
    std::mem_fun(
        &std::ctype<char>::tolower),
    &std::use_facet<std::ctype<char> >(
        std::locale()));
```



gives you a functor that uses the active locale to convert characters to lowercase, which you can then use via `std::transform` to generate lower-case strings:

```
std::string left = "fOo";  
transform(left.begin(), left.end(), left.begin(), tolower);
```

This also works for `wchar_t` based strings.

Share Improve this answer

answered Sep 29, 2015 at 0:25

Follow



[Simon Richter](#)

29.5k ● 1 ● 47 ● 67

Definitely better than all the rest using lower/upper without considering the i18n issues – [nimish](#) Mar 25, 2023 at 3:15



4



I've had good experience using the [International Components for Unicode libraries](#) - they're extremely powerful, and provide methods for conversion, locale support, date and time rendering, case mapping (which you don't seem to want), and [collation](#), which includes case- and accent-insensitive comparison (and more). I've only used the C++ version of the libraries, but they appear to have a Java version as well.

Methods exist to perform normalized compares as referred to by @Coincoin, and can even account for locale - for example (and this a sorting example, not strictly equality), traditionally in Spanish (in Spain), the

letter combination "ll" sorts between "l" and "m", so "lz" < "ll" < "ma".

Share Improve this answer

edited Aug 26, 2008 at 11:02

Follow

answered Aug 14, 2008 at 20:29



Blair Conrad

241k ● 25 ● 136 ● 112



4

Just use `strcmp()` for case sensitive and `strncmpi()` or `stricmp()` for case insensitive comparison. Which are both in the header file `<string.h>`



format:



```
int strcmp(const char*, const char*);    //for case sen
int strncmpi(const char*, const char*);  //for case ins
```

Usage:

```
string a="apple",b="ApPlE",c="ball";
if(strncmpi(a.c_str(),b.c_str())==0)      //(if it is a
    cout<<a<<" and "<<b<<" are the same"<<"\n";
if(strncmpi(a.c_str(),b.c_str())<0)
    cout<<a[0]<<" comes before ball "<<b[0]<<" , so "<<
```

Output

apple and ApPlE are the same

a comes before b, so apple comes before ball

Share Improve this answer

edited Jul 26, 2013 at 15:15

Follow

answered Jul 25, 2013 at 18:47



reubenjohn

1,409 ● 1 ● 20 ● 44

-
- 2 Downvote because this is hardly a C++ way of doing things.
– [Thomas Daugaard](#) Jul 30, 2013 at 9:09
-

This is the c++ convention at my university but i will keep it in mind when posting here – [reubenjohn](#) Aug 13, 2013 at 18:04



-
- 4 strcmp is a Microsoft extension AFAIK. BSD seems to have strcasecmp() instead. – [uliwitness](#) Jan 22, 2014 at 13:30
-



A simple way to compare two string in c++ (tested for windows) is using **_stricmp**

3



```
// Case insensitive (could use equivalent _stricmp)
result = _stricmp( string1, string2 );
```



If you are looking to use with std::string, an example:



```
std::string s1 = string("Hello");
if ( _stricmp(s1.c_str(), "HELLO") == 0 )
    std::cout << "The string are equals.";
```

For more information here: <https://msdn.microsoft.com/it-it/library/e0z9k731.aspx>

Share Improve this answer

edited Apr 26, 2018 at 21:42

Follow



eyllanesc

243k ● 19 ● 196 ● 274

answered Apr 18, 2018 at 9:33



DAme

727 ● 8 ● 21

-
- 1 It's worth reading stackoverflow.com/a/12414441/95309 in addition to this answer, as it's a) a C function, and b) supposedly not portable. – [Claus Jørgensen](#) Aug 23, 2018 at 12:18

what `#include` do we need to make this work? – [ekkis](#) Jun 22, 2019 at 21:37

-
- 1 @ekkis to use `_stricmp` you have to include `<string.h>` as you can read here: learn.microsoft.com/en-us/cpp/c-runtime-library/reference/... – [DAme](#) Jul 1, 2019 at 7:47

-
- 1 Nice try microsoft! – [AdrianTut](#) Mar 25, 2022 at 10:38
-



2



Just a note on whatever method you finally choose, if that method happens to include the use of `strcmp` that some answers suggest:

`strcmp` doesn't work with Unicode data in general. In general, it doesn't even work with byte-based Unicode encodings, such as utf-8, since `strcmp` only makes byte-per-byte comparisons and Unicode code points encoded in utf-8 can take more than 1 byte. The only specific Unicode case `strcmp` properly handle is when a string encoded with a byte-based encoding contains only code

points below U+00FF - then the byte-per-byte comparison is enough.

Share Improve this answer

answered Nov 25, 2008 at 7:26

Follow



Johann Gerell

25.5k ● 11 ● 76 ● 125



2

Looks like above solutions aren't using compare method and implementing total again so here is my solution and hope it works for you (It's working fine).



```
#include<iostream>
#include<cstring>
#include<cmath>
using namespace std;
string tolow(string a)
{
    for(unsigned int i=0;i<a.length();i++)
    {
        a[i]=tolower(a[i]);
    }
    return a;
}
int main()
{
    string str1,str2;
    cin>>str1>>str2;
    int temp=tolow(str1).compare(tolow(str2));
    if(temp>0)
        cout<<1;
    else if(temp==0)
        cout<<0;
    else
        cout<<-1;
}
```

Share Improve this answer

answered Aug 26, 2017 at 19:57

Follow



Jagadeesh
Pulamarasetti

483 ● 4 ● 8



As of early 2013, the ICU project, maintained by IBM, is a pretty good answer to this.

1

<http://site.icu-project.org/>



ICU is a "complete, portable Unicode library that closely tracks industry standards." For the specific problem of string comparison, the Collation object does what you want.



The Mozilla Project adopted ICU for internationalization in Firefox in mid-2012; you can track the engineering discussion, including issues of build systems and data file size, here:

- <https://groups.google.com/forum/#!topic/mozilla.dev.platform/sVVpS2sKODw>
- https://bugzilla.mozilla.org/show_bug.cgi?id=724529 (tracker)
- https://bugzilla.mozilla.org/show_bug.cgi?id=724531 (build system)

Share Improve this answer

answered Apr 1, 2013 at 17:58

Follow



michaelhanson

369 ● 2 ● 5



1



If you have to compare a source string more often with other strings one elegant solution is to use regex.

```
std::wstring first = L"Test";  
std::wstring second = L"TEST";  
  
std::wregex pattern(first, std::wregex::icase);  
bool isEqual = std::regex_match(second, pattern);
```

Share Improve this answer

edited Sep 30, 2015 at 12:47

Follow

answered Mar 6, 2015 at 13:55



smibe

159 ● 11

Tried this but compile error: `error: conversion from 'const char [5]' to non-scalar type 'std::wstring {aka std::basic_string<wchar_t>}' requested`

– [Deqing](#) May 15, 2015 at 5:18

bad idea. It is the worst solution. – [Behrouz.M](#) Jun 1, 2015 at 13:05

This isn't a good solution, but even if you wanted to use it, you need an L in front of your widestring constants, eg L"TEST" – [celticminstrel](#) Jun 21, 2015 at 2:27

Would be nice if someone could explain why it is the worst solution. Because of performance issues? Creating the regex is expensive, but afterwards the comparison should be really fast. – [smibe](#) Sep 30, 2015 at 12:49

it's usable and portable, the major problem is that first can't contain any characters that regex uses. It can't be used as a

general string compare because of that. It will also be slower, there is a flag to make it work the way smibe says but still can't be used as a general function. – Ben Aug 16, 2016 at 21:37



If you don't want to use **Boost library** then here is solution to it using only C++ standard io header.

0



```
#include <iostream>
#include <cctype>
#include <algorithm>
#include <stdexcept>
#include <cassert>

struct iequal
{
    bool operator()(int c1, int c2) const
    {
        return std::toupper(c1) == std::toupper(c2);
    }
};

bool iequal(const std::string& str1, const std::string& str2)
{
    if (str1.empty() || str2.empty())
    {
        return str1.empty() && str2.empty();
    }
    return std::equal(str1.begin(), str1.end(), str2.begin(), str2.end(), iequal);
}

void runTests()
{
    assert(iequal("HELLO", "hello") == true);
    assert(iequal("HELLO", "") == false);
    assert(iequal("", "hello") == false);
    assert(iequal("", "") == true);
    std::cout << "All tests passed!" << std::endl;
}
```

```
int main(void)
{
    try
    {
        runTests();
    }
    catch (const std::exception& e)
    {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

Share Improve this answer

edited Sep 14, 2023 at 12:19

Follow

answered Apr 26, 2018 at 21:40



HeavenHM

992 ● 1 ● 14 ● 22

I believe `std::toupper` is in `#include <cctype>`, you might need to include it. – [David Ledger](#) Dec 2, 2018 at 12:00

If you will use global version like this `::toupper` then you might not need to include `<cctype>` because there are two versions `c` version and `c++` version with locale i guess. So better to use global version `::toupper()` – [HeavenHM](#) Dec 2, 2018 at 13:52

this solution fails when one of the strings is empty: `""` -- it returns true in that case when it should return false – [ekkis](#) Jun 22, 2019 at 21:35 ✎

@ekkis Now i have updated it with Unit tests now you can check – [HeavenHM](#) Sep 14, 2023 at 12:19



-2



```

bool insensitive_c_compare(char A, char B){
    static char mid_c = ('Z' + 'a') / 2 + 'Z';
    static char up2lo = 'A' - 'a'; /// the offset between

    if ('a' >= A and A >= 'z' or 'A' >= A and 'Z' >= A)
        if ('a' >= B and B >= 'z' or 'A' >= B and 'Z' >= B)
            /// check that the character is infact a letter
            /// (trying to turn a 3 into an E would not be p
            {
                if (A > mid_c and B > mid_c or A < mid_c and B
                {
                    return A == B;
                }
                else
                {
                    if (A > mid_c)
                        A = A - 'a' + 'A';
                    if (B > mid_c)/// convert all uppercase lett
                        B = B - 'a' + 'A';
                    /// this could be changed to B = B + up2lo;
                    return A == B;
                }
            }
        }
    }
}

```

this could probably be made much more efficient, but here is a bulky version with all its bits bare.

not all that portable, but works well with whatever is on my computer (no idea, I am of pictures not words)

Share Improve this answer

answered Mar 5, 2015 at 2:40

Follow



[user4578093](#)

229 ● 1 ● 3 ● 10

This is not Unicode support which is what the question asked. – [Behrouz.M](#) Jun 1, 2015 at 13:06

This doesn't support non-english character sets.

– [Robert Andrzejuk](#) Apr 29, 2018 at 15:02



-4



An easy way to compare strings that are only different by lowercase and capitalized characters is to do an ascii comparison. All capital and lowercase letters differ by 32 bits in the ascii table, using this information we have the following...



```
for( int i = 0; i < string2.length(); i++)
{
    if (string1[i] == string2[i] || int(string1[i])
||int(string1[i]) == int(string2[i])-32)
    {
        count++;
        continue;
    }
    else
    {
        break;
    }
    if(count == string2.length())
    {
        //then we have a match
    }
}
```

Share Improve this answer

Follow

edited May 12, 2015 at 14:28



[HaveNoDisplayName](#)

8,487 ● 106 ● 40 ● 50

answered May 12, 2015 at 14:17



[Craig Stoddard](#)

1

-
- 3 According to this, "++j" will be found equal to "KKJ", and "1234" will be found equal to "QRST". I doubt that's something anyone wants. – [celticminstrel](#) Jun 21, 2015 at 2:24
-