How to do case-insensitive string comparison
Matt Austern

If you've ever written a program that uses strings (and who hasn't?), chances are you've sometimes needed to treat two strings that differ only by case as if they were identical. That is, you've needed comparisons-equality, less than, substring matches, sorting-to disregard case. And, indeed, one of the most frequently asked questions about the Standard C++ Library is how to make strings case insensitive. This question has been answered many times. Many of the answers are wrong.

First of all, let's dispose of the idea that you should be looking for a way to write a case-insensitive string class. Yes, it's technically possible, more or less. The Standard Library type `std::string` is really just an alias for the template `std::basic_string<char, std::char_traits<char>, std::allocator<char> >`. It uses the traits parameter for all comparisons, so, by providing a traits parameter with equality and less than redefined appropriately, you can instantiate `basic_string` in such a way so that the < and == operators are case insensitive. You can do it, but it isn't worth the trouble.
You won't be able to do I/O, at least not without a lot of pain. The I/O classes in the Standard Library, like `std::basic_istream` and `std::basic_ostream`, are templatized on character type and traits just like `std::basic_string` is. (Again, `std::ostream` is just an alias for `std::basic_ostream<char, char_traits<char> >`.) The traits parameters have to match. If you're using `std::basic_string<char, my_traits_class>` for your strings, you'll have to use `std::basic_ostream<char, my_traits_class>` for your stream output. You won't be able use ordinary stream objects like `cin` and `cout`.

- *Case insensitivity isn't about an object, it's about how you use an object.* You might very well need to treat a string as case-sensitive in some contexts and case-insensitive in others. (Perhaps depending on a user-controlled option.) Defining separate types for those two uses puts an artificial barrier between them.
- *It doesn't quite fit.* Like all traits classes[1], `char_traits` is small, simple, and stateless. As we'll see later in this column, proper case-insensitive comparisons are none of those things.
- *It isn't enough.* Even if all of `basic_string`'s own member functions were case insensitive, that still wouldn't help when you need to use nonmember generic algorithms like `std::search` and `std::find_end`. It also wouldn't help if you decided, for reasons of efficiency, to change from a container of `basic_string` objects to a string table.

A better solution, one that fits more naturally into the design of the Standard Library, is to ask for case-insensitive comparison when that's what you need. Don't bother with member functions like `string::find_first_of` and `string::rfind`; all of them duplicate functionality that's already there in nonmember generic algorithms. The generic algorithms, meanwhile, are

[1]See Andrei Alexandrescu's column in the April issue.

flexible enough to accommodate case-insensitive strings. If you need to sort a collection of strings in case-insensitive order, for example, all you have to do is provide the appropriate comparison function object:

```
std::sort(C.begin(), C.end(), compare_without_case);
```

The remainder of this column will be about how to write that function object.

A first attempt

There's more than one way to alphabetize words. The next time you're in a bookstore, check how the authors' names are arranged: does Mary McCarthy come before Bernard Malamud, or after? (It's a matter of convention, and I've seen it done both ways.) The simplest kind of string comparison, though, is the one we all learned in elementary school: lexicographic or "dictionary order" comparison, where we build up string comparison from character-by-character comparison.

Lexicographic comparison may not be suitable for specialized applications (no single method is; a library might well sort personal names and place names differently), but it's suitable much of the time and it's what string comparison means in C++ by default. Strings are sequences of characters, and if `x` and `y` are of type `std::string`, the expression `x < y` is equivalent to the expression

```
std::lexicographical_compare(x.begin(), x.end(),
                             y.begin(), y.end()).
```

In this expression `lexicographical_compare` compares individual characters using `operator<`, but there's also a version of `lexicographical_compare` that lets you choose your own method of comparing characters. That other version takes five arguments, not four; the last argument is a function object, a Binary Predicate that determines which of two characters should precede the other. All we need in order to use `lexicographical_compare` for case-insensitive string comparison, then, is to combine it with a function object that compares characters without regard to case.

The general idea behind case-insensitive comparison of characters is to convert both characters to upper-case and compare the results. Here's the obvious translation of that idea into a C++ function object, using a well known function from the standard C library:

```
struct lt_nocase
  : public std::binary_function<char, char, bool> {
  bool operator()(char x, char y) const {
    return toupper(static_cast<unsigned char>(x)) <
           toupper(static_cast<unsigned char>(y));
  }
};
```

"For every complex problem there is a solution that is simple, neat, and wrong." People who write books about C++ are fond of this class, because it makes a nice, simple example. I'm as guilty as anyone else; I use it in my book a half dozen times. It's almost right, but that's not good enough. The problem is subtle.

Here's one example where you can begin to see the problem:

```
int main()
{
  const char* s1 = "GEW\334RZTRAMINER";
  const char* s2 = "gew\374rztraminer";
  printf("s1 = %s,  s2 = %s\n", s1, s2);

  printf("s1 < s2: %s\n",
         std::lexicographical_compare(s1, s1 + 14,
                                      s2, s2 + 14,
                                      lt_nocase())
    ? "true" : "false");
}
```

You should try this out on your system. On my system (a Silicon Graphics O2 running IRIX 6.5), here's the output:

```
s1 = GEWÜRZTRAMINER,  s2 = gewürztraminer
s1 < s2: true
```

Hm, how odd. If you're doing a case-insensitive comparison, shouldn't "`gewürztraminer`" and "`GEWÜRZTRAMINER`" be the same? And now a slight variation: if you insert the line

```
setlocale(LC_ALL, "de");
```

just before the `printf` statement, suddenly the output changes:

```
s1 = GEWÜRZTRAMINER,  s2 = gewürztraminer
s1 < s2: false
```

Case-insensitive string comparison is more complicated than it looks. This seemingly innocent program depends crucially on something most of us would prefer to ignore: locales.

Locales

A `char` is really nothing more than a small integer. We can choose to interpret a small integer as a character, but there's nothing universal about that interpretation. Should some particular number be interpreted as a letter, a punctuation mark, or a nonprinting control character?

There's no single right answer, and it doesn't even make a difference as far as the core C and C++ languages are concerned. A few library function need to make those distinctions: `isalpha`, for example, which determines whether a character is a letter, and `toupper`, which converts lowercase letters to uppercase and does nothing to uppercase letters or to characters that aren't letters. All of that depends on local cultural and linguistic conventions: distinguishing between letters and nonletters means one thing in English, another thing in Swedish. Conversion from lower to upper case means something different in the Roman alphabet than in the Cyrillic, and means nothing at all in Hebrew.

By default the character manipulation functions work with a character set that's appropriate for simple English text. The character `'\374'` isn't affected by `toupper` because it isn't a letter; it may look like ü when it's printed on some systems, but that's irrelevant for a C library routine that's operating on English text. There is no ü character in the ASCII character set. The line

```
      setlocale(LC_ALL, "de");
```

tells the C library to start operating in accordance with German conventions. (At least it does on IRIX. Locale names are not standardized.) There is a character ü in German, so `toupper` changes ü to Ü.

If this doesn't make you nervous, it should. While `toupper` may look like a simple function that takes one argument, it also depends on a global variable-worse, a hidden global variable. This causes all of the usual difficulties: a function that uses `toupper` potentially depends on every other function in the entire program.

This can be disastrous if you're using `toupper` for case-insensitive string comparison. What happens if you've got an algorithm that depends on a list being sorted (binary search, say), and then a new locale causes the sort order to change out from under it? Code like this isn't reusable; it's barely usable. You can't use it in libraries-libraries get used in all sorts of programs, not just programs that never call `setlocale`. You might be able to get away with using it in a large program, but you'll have a maintenance problem: maybe you can prove that no other module ever calls `setlocale`, but can you prove that no other module in next year's version of the program will call `setlocale`?

There's no good solution to this problem in C. The C library has a single global locale, and that's that. There is a solution in C++.

Locales in C++

A locale in the C++ Standard Library isn't global data buried deep within the library implementation. It's an object of type `std::locale`, and you can create it and pass it to functions just like any other object. You can create a locale object that represents the usual locale, for example, by writing

```
      std::locale L = std::locale::classic();,
```

or you can create a German locale by writing

```
      std::locale L("de");.
```

(As in the C library, names of locales aren't standardized. Check your implementation's documentation to find out what named locales are available.)

Locales in C++ are divided into *facets*, each of which handles a different aspect of internationalization, and the function `std::use_facet` extracts a specific facet from a locale object[2]. The `ctype` facet handles character classification, including case conversion. Finally, then, if `c1` and `c2` are of type `char`, this fragment will compare them in a case-

[2]Warning: `use_facet` is a function template whose template parameter appears only in the return type, not in any of the arguments. Calling it uses a language feature called *explicit template argument specification*, and some C++ compilers don't support that feature yet. If you're using a compiler that doesn't support it, your library implementor may have provided a workaround so that you can call `use_facet` some other way.

insensitive manner that's appropriate to the locale `L`.

```
      const std::ctype<char>& ct
         = std::use_facet<std::ctype<char> >(L);
      bool result = ct.toupper(c1) < ct.toupper(c2);
```

There's also a special abbreviation: you can write

```
      std::toupper(c, L),
```

which (if `c` is of type `char`) means the same thing as

```
      std::use_facet<std::ctype<char> >(L).toupper(c).
```

It's worth minimizing the number of times you call `use_facet`, though, because it can be fairly expensive.

A digression: another facet

If you're already familiar with locales in C++, you may have thought of another way to perform string comparisons: The `collate` facet exists precisely to encapsulate details of sorting, and it has a member function with an interface much like that of the C library function `strcmp`. There's even a little convenience feature: if `L` is a locale object, you can compare two strings by writing `L(x, y)` instead of going through the nuisance of calling `use_facet` and then invoking a `collate` member function.

The "classic" locale has a `collate` facet that performs lexicographical comparison, just like `string`'s `operator<` does, but other locales perform whatever kind of comparison is appropriate. If your system happens to come with a locale that performs case-insensitive comparisons for whatever languages you're interested in, you can just use it.

Unfortunately, this piece of advice, true as it may be, isn't much help for those of us who don't have such systems. Perhaps someday a set of such locales may be standardized, but right now they aren't. If someone hasn't already written a case-insensitive comparison function for you, you'll have to write it yourself.

Case-insensitive string comparison

Using `ctype`, it's straightforward to build case-insensitive string comparison out of case-insensitive character comparison. This version isn't optimal, but at least it's correct. It uses essentially the same technique as before: compare two strings using `lexicographical_compare`, and compare two characters by converting both of them to uppercase. The time, though, we're being careful to use a locale object instead of a global variable. (As an aside, converting both characters to uppercase might not always give the same results as converting both characters to lowercase: There's no guarantee that the operations are inverses. In French, for example, it's customary to omit accent marks on uppercase characters. In a French locale it might be reasonable for `toupper` to be a lossy conversion; it might convert both `'é'` and `'e'` to the same uppercase character, `'E'`. In such a locale, then, a case-insensitive comparison using `toupper` will say that `'é'` and `'E'` are equivalent characters while a case-insensitive comparison using `tolower` will say that they aren't. Which is the right answer? Probably the former, but it depends on the language, on local customs, and on your application.)

```
struct lt_str_1 : public
  std::binary_function<std::string, std::string, bool>
{
  struct lt_char {
    const std::ctype<char>& ct;
    lt_char(const std::ctype<char>& c) : ct(c) {}
    bool operator()(char x, char y) const {
      return ct.toupper(x) < ct.toupper(y);
    }
  };

  std::locale loc;
  const std::ctype<char>& ct;

  lt_str_1(const std::locale& L = std::locale::classic())
    : loc(L), ct(std::use_facet<std::ctype<char> >(loc)) {}

  bool operator()(const std::string& x,
                  const std::string& y) const {
    return std::lexicographical_compare(x.begin(), x.end(),
                                        y.begin(), y.end(),
                                        lt_char(ct));
  }
};
```

This isn't quite optimal yet; it's slower than it ought to be. The problem is annoying and technical: we're calling `toupper` in a loop, and the C++ standard requires `toupper` to make a virtual function call. Some optimizers may be smart enough to move the virtual function overhead out of the loop, but most aren't. Virtual function calls in loops should be avoided.

In this case, avoiding it isn't completely straightforward. It's tempting to think that the right answer is another one of `ctype`'s member functions,

```
const char* ctype<char>::toupper(char* f, char* l) const,
```

which changes the case of the characters in the range `[f, l)`. Unfortunately, this isn't quite the right interface for our purpose. Using it to compare two strings would require copying both strings to buffers and then converting the buffers to uppercase. Where do those buffers come from? They can't be fixed size arrays (how large is large enough?), but dynamic arrays would require an expensive memory allocation.

An alternative solution is to do the case conversion once for every character, and cache the result. This isn't a fully general solution-it would be completely unworkable, for example, if you were working with 32-bit UCS-4 characters. If you're working with `char`, though (8 bits on most systems), it's not so unreasonable to maintain 256 bytes of case conversion information in the comparison function object.

```
struct lt_str_2 : public
```

```
  std::binary_function<std::string, std::string, bool>
{
  struct lt_char {
    const char* tab;
    lt_char(const char* t) : tab(t) { }
    bool operator()(char x, char y) const {
      return tab[x - CHAR_MIN] < tab[y - CHAR_MIN];
    }
  };

  char tab[CHAR_MAX - CHAR_MIN + 1];

  lt_str_2(const std::locale& L = std::locale::classic()) {
    const std::ctype<char>& ct
      = std::use_facet<std::ctype<char> >(L);
    for (int i = CHAR_MIN; i <= CHAR_MAX; ++i)
      tab[i - CHAR_MIN] = (char) i;
    ct.toupper(tab, tab + (CHAR_MAX - CHAR_MIN + 1));
  }

  bool operator()(const std::string& x,
                  const std::string& y) const {
    return std::lexicographical_compare(x.begin(), x.end(),
                                        y.begin(), y.end(),
                                        lt_char(tab));
  }
};
```

As you can see, `lt_str_1` and `lt_str_2` aren't very different. The former has a character-comparison function object that uses a `ctype` facet directly, and the latter a character-comparison function object that uses a table of precomputed uppercase conversions. This might be slower if you're creating an `lt_str_2` function object, using it compare a few short strings, and then throwing it away. For any substantial use, though, `lt_str_2` will be noticeably faster than `lt_str_1`. On my system the difference was more than a factor of two: it took 0.86 seconds to sort a list of 23,791 words with `lt_str_1`, and 0.4 with `lt_str_2`.

What have we learned from all of this?

- A case-insensitive string class is the wrong level of abstraction. Generic algorithms in the C++ standard library are parameterized by policy, and you should exploit that fact.
- Lexicographical string comparisons are built out of character comparisons. Once you've got a case-insensitive character comparison function object, the problem is solved. (And you can reuse that function object for comparisons of other kinds of character sequences, like `vector<char>`, or string tables, or ordinary C strings.)
- Case-insensitive character comparison is harder than it looks. It's meaningless except in the context of a particular locale, so a character comparison function object needs to store

locale information. If speed is a concern, you should write the function object to avoid repeated calls of expensive facet operations.

Correct case-insensitive comparison takes a fair amount of machinery, but you only have to write it once. You probably don't want to think about locales; most people don't. (Who wanted to think about Y2K bugs in 1990?) You stand a better chance of being able to ignore locales if you get locale-dependent code right, though, than if you write code that glosses over the dependencies.