Advanced Programming in C++ Exercise – Character traits

Character sets differ in various respects, which raises a number of questions concerning representation and how operations on character and character arrays are to be performed (as efficient as possible):

- What integer type to use for representing all characters, including the end-of-file character?
- How convert between characters and their integer representation?
- How assign a single character to another?
- How initialize a character array or how assign a character array to another?
- How compare one character with another?
- How compare one character array with another?
- How compute the length of a character array?
- How copy the content of one character array into another? If their memory don't overlap? If their memory may overlap?

Such characteristics are called *character traits* in the standard. The standard template char_traits<charT>, declared in <string>, defines the interface for standard character traits. Since it is not possible to define an implementation of char_traits<charT> that would work for all possible character types charT, the standard only declares the template:

template<class charT> struct char_traits;

(There may be a definition, for which the member functions are not defined.) The declaration is provided in header <string> as a basis for explicit specializations, of which there are two given in the standard:

```
template<> struct char_traits<char>;
template<> struct char_traits<wchar_t>;
```

The definition of char_traits<char> (and for char_traits<char>) provided by the standard is (also in <string>):

```
template<>
struct char traits<char>
ł
  typedef charT char_type;
typedef int int_type;
  typedef streampos pos_type;
  typedef streamoff off_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char type& c1, const char type& c2);
  static int compare(const char type* s1, const char type* s2, size t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s, size_t n,
                                const char_type& a);
  static char_type* move(char_type* s1, const char_type* s2, size_t n);
  static char_type* copy(char_type* s1, const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n, char_type a);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1, const int_type& c2);
  static int type eof();
  static int_type not_eof(const int_type& c);
};
```

The requirements for the traits operations are:

assign(c1, c2)	Yields: nothing. Assigns c1=c2. Complexity: constant. <i>Note</i> : for char and wchar_t specializations, assign is defined identical to the builtin operator =.
eq(c1, c2)	Return type: bool . Yields: whether c1 is to be treated as equal to c2. <i>Note</i> : for char and wchar_t specializations, eq is defined identical to the builtin operator ==. Complexity: constant.
lt(c1, c2)	Return type: bool . Yields: whether c1 is to be treated as less than c2. <i>Note</i> : for char and wchar_t specializations, lt is defined identical to the builtin operator <. Complexity: constant.
compare(s1, s2, n)	Return type: int . Yields: 0 if for each i in $[0, n[, eq(s1[i], s2[i]) is true; else, a negative value if, for some j in [0, n[, lt(s1[j], s2[j]) is true and for each i in [0, j[, lt(s1[i] s2[i]) is true, else a positive value. Complexity: linear.$
length(s)	Return type: size_t. Yields: the smallest i such that eq(s[i], charT()) is true. Complexity: linear.
find(s, n, c)	Return type: const char_type*. Yields: the smallest p in [s, s+n[such that eq(*p, c) is true , zero otherwise. Complexity: linear.
move(s1, s2, n)	Return type: char_type*. Yields s1. For each i in [0, n[, performs assign(s1[i], s2[i]). Copies correctly even if s2 is in [s1, s1+n[. Complexity: linear.
copy(s1, s2, n)	Precondition: s2 not in [s1, s1+n[. Yields s1. For each i in [0, n[, performs assign(s1[i], s2[i]). Complexity: linear.
assign(s, n, a)	Return type: char_type*. Yields: s. For each i in [0, n[, performs assign(s[i], a). Complexity: linear.
not_eof(c)	Return type: int_type. Yields: c if eq_int_type(c, eof()) is false , otherwise a value e such that eq_int_type(e, eof()) is false . Complexity: constant. <i>Note</i> : for char and wchar_t specializations, e could be zero.
to_char_type(c)	Return type: char_type. Yields: if for some c2, eq_int_type(c, to_int_type(c2)) is true , c2; else some unspecified value. Complexity: constant. <i>Note</i> : f for char and wchar_t specializations, yield c converted to char_type.
to_int_type(c)	Return type: int_type. Yields: some value e, constrained by definitions of to_char_type and eq_int_type. Complexity: constant. <i>Note</i> : for char and wchar_t specializations, make sure that the byte 0xff and the end-of-file symbol 0xffffffff don't both end up as 0xffffffff (by clever type conversions).
eq_int_type(c1, c2)	Return type: bool . Yields: for all c and d, eq(c, d) is equal to eq_int_type(to_int_type(c), to_int_type(d)); otherwise, yields true if c1 and c2 are both copies of eof(); otherwise, yields false if one of c1 and c2 is a copy of eof() and the other is not; otherwise the value is unspecified. Complexity: constant. <i>Note</i> : for the char and wchar_t specializations, eq_int_type may be defined as the builtin operator ==.
eof()	Return type: int_type. Yields: a value e such that eq_int_type(e, to_int_type(c)) is false for all values c. Complexity: constant. <i>Note</i> : For char specialization EOF is returned.

Note: for **char** and **wchar_t** specializations, standard string functions from <cstring> may be used to implement compare, length, find, move, copy and assign (three parameter version). This allows for both simple and efficient implementations, especially since the memory operations (mem*, wmem*) can be used in most cases.

Quite a few standard templates, such as basic_string and basic stream classes, e.g. basic_istream, have a template parameter for a character type and a character traits:

Normally we use type string for strings, which is defined as a specialization of basic_string for **char**, and for wide strings we use wstring, which is defined as a specialization for **wchar_t**:

typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;

By default these specializations use char_traits<char> and char_traits<wchar_t>, respectively.

Here are two examples from the streams library, where char_traits are used, by default in the first case:

typedef basic_istream<char> istream;

typedef fpos<char_traits<char>::state_type> streampos;

In this exercise you will experiment some with *character traits*. Exercises are given on the next page.

1. Implement char_traits<char> specialization

This specialization of char_traits<charT> is given in the standard, but, as an exercise, you can define your own implementation of the member functions. Some hints are given in the specifications given above.

2. Case-insensitive string type

Define a string type, ci_string, for **char** which is not case-sensitive. Do this by defining a character traits class ci_char_traits and specialize basic_string for **char** and ci_char_traits.

Write a test program to test ci_string and, indirectly or explicitly, ci_char_traits. It shall also be possible to read ci_string values from istreams using operator >>, and to write ci_string values to ostreams using operator <<.

Hint: Most of the functions defined in std::char_traits<char> are case-insensitive ...

3. Swedish string type

Define a string type, sv_string, for **char** which will work properly for lexicographical comparison also if strings contain Swedish letters. Do this by defining a character traits sv_char_traits and specialize basic_string for **char** and sv_char_traits.

Write a test program to test ci_string and, indirectly or explicitly, ci_char_traits . It shall also be possible to read ci_string values from istreams using operator >>, and to write ci_string values to ostreams using operator <<.

Of course, you may do this for some other language than Swedish, if you prefer.

Note: The Swedish alphabet comprise the letters a-z and å, ä, ö, in that order.