Searching in strings

The **find** family of **string** member functions allows you to locate a character or group of characters within a given string. Here are the members of the **find** family and their general usage:

string find member function	What/how it finds
find( )	Searches a string for a specified character or group of characters and returns the starting position of the first occurrence found or **npos** (this member datum holds the current actual length of the string which is being searched) if no match is found.
find_first_of( )	Searches a target string and returns the position of the first match of *any* character in a specified group. If no match is found, it returns **npos**.
find_last_of( )	Searches a target string and returns the position of the last match of *any* character in a specified group. If no match is found, it returns **npos**.
find_first_not_of( )	Searches a target string and returns the position of the first element that *doesn't* match of *any* character in a specified group. If no such element is found, it returns **npos**.
find_last_not_of( )	Searches a target string and returns the position of the element with the largest

	subscript that *doesn't* match of *any* character in a specified group. If no such element is found, it returns **npos**.
rfind( )	Searches a string from end to beginning for a specified character or group of characters and returns the starting position of the match if one is found. If no match is found, it returns **npos**.

String searching member functions and their general uses

The simplest use of **find( )** searches for one or more characters in a **string**. This overloaded version of **find( )** takes a parameter that specifies the character(s) for which to search, and optionally one that tells it where in the string to begin searching for the occurrence of a substring. (The default position at which to begin searching is 0.) By setting the call to **find** inside a loop, you can easily move through a string, repeating a search in order to find all of the occurrences of a given character or group of characters within the string.

Notice that we define the string object **sieveChars** using a constructor idiom which sets the initial size of the character array and writes the value 'P' to each of its member.

```
//: C17:Sieve.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  // Create a 50 char string and set each
  // element to 'P' for Prime
  string sieveChars(50, 'P');
  // By definition neither 0 nor 1 is prime.
  // Change these elements to "N" for Not Prime
  sieveChars.replace(0, 2, "NN");
  // Walk through the array:
  for(int i = 2;
    i <= (sieveChars.size() / 2) - 1; i++)
    // Find all the factors:
    for(int factor = 2;
      factor * i < sieveChars.size();factor++)
      sieveChars[factor * i] = 'N';

  cout << "Prime:" << endl;
  // Return the index of the first 'P' element:
  int j = sieveChars.find('P');
  // While not at the end of the string:
  while(j != sieveChars.npos) {
```

```
    // If the element is P, the index is a prime
    cout << j << " ";
    // Move past the last prime
    j++;
    // Find the next prime
    j = sieveChars.find('P', j);
  }
  cout << "\n Not prime:" << endl;
  // Find the first element value not equal P:
  j = sieveChars.find_first_not_of('P');
  while(j != sieveChars.npos) {
    cout << j << " ";
    j++;
    j = sieveChars.find_first_not_of('P', j);
  }

} ///:~
```

The output from **Sieve.cpp** looks like this:

```
Prime:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
Not prime:
0 1 4 6 8 9 10 12 14 15 16 18 20 21 22
24 25 26 27 28 30 32 33 34 35 36 38 39
40 42 44 45 46 48 49
```

find( ) allows you to walk forward through a **string**, detecting multiple occurrences of a character or group of characters, while **find_first_not_of( )** allows you to test for the absence of a character or group.

The **find** member is also useful for detecting the occurrence of a sequence of characters in a **string**:

```
//: C17:Find.cpp
// Find a group of characters in a string
#include <string>
#include <iostream>
using namespace std;

int main() {
  string chooseOne("Eenie, meenie, miney, mo");
  int i = chooseOne.find("een");
  while(i != string::npos) {
    cout << i << endl;
    i++;
    i = chooseOne.find("een", i);
  }

} ///:~
```

Find.cpp produces a single line of output :

```
8
```

This tells us that the first 'e' of the search group "een" was found in the word "meenie," and is the eighth element in the string. Notice that **find** passed over the "Een" group of characters in the word "Eenie". The **find** member function performs a *case sensitive* search.

There are no functions in the **string** class to change the case of a string, but these functions can be easily created using the Standard C library functions **toupper( )** and **tolower( )**, which change the case of one character at a time. A few small changes will make **Find.cpp** perform a case insensitive search:

```
//: C17:NewFind.cpp
#include <string>
#include <iostream>
using namespace std;

// Make an uppercase copy of s:
string upperCase(string& s) {
  char* buf = new char[s.length()];
  s.copy(buf, s.length());
  for(int i = 0; i < s.length(); i++)
    buf[i] = toupper(buf[i]);
  string r(buf, s.length());
  delete buf;
  return r;
}

// Make a lowercase copy of s:
string lowerCase(string& s) {
  char* buf = new char[s.length()];
  s.copy(buf, s.length());
  for(int i = 0; i < s.length(); i++)
    buf[i] = tolower(buf[i]);
  string r(buf, s.length());
  delete buf;
  return r;
}

int main() {
  string chooseOne("Eenie, meenie, miney, mo");
  cout << chooseOne << endl;
  cout << upperCase(chooseOne) << endl;
  cout << lowerCase(chooseOne) << endl;
  // Case sensitive search
  int i = chooseOne.find("een");
  while(i != string::npos) {
    cout << i << endl;
    i++;
    i = chooseOne.find("een", i);
  }
  // Search lowercase:
  string lcase = lowerCase(chooseOne);
  cout << lcase << endl;
  i = lcase.find("een");
  while(i != lcase.npos) {
    cout << i << endl;
    i++;
```

```
    i = lcase.find("een", i);
  }
  // Search uppercase:
  string ucase = upperCase(chooseOne);
  cout << ucase << endl;
  i = ucase.find("EEN");
  while(i != ucase.npos) {
    cout << i << endl;
    i++;
    i = ucase.find("EEN", i);
  }
} ///:~
```

Both the **upperCase( )** and **lowerCase( )** functions follow the same form: they allocate storage to hold the data in the argument **string**, copy the data and change the case. Then they create a new **string** with the new data, release the buffer and return the result **string**. The **c_str( )** function cannot be used to produce a pointer to directly manipulate the data in the **string** because **c_str( )** returns a pointer to **const**. That is, you're not allowed to manipulate **string** data with a pointer, only with member functions. If you need to use the more primitive **char** array manipulation, you should use the technique shown above.

The output looks like this:

```
Eenie, meenie, miney, mo
eenie, meenie, miney, mo
EENIE, MEENIE, MINEY, MO
8
eenie, meenie, miney, mo
0
8
EENIE, MEENIE, MINEY, MO
0

8
```

The case insensitive searches found both occurrences on the "een" group.

NewFind.cpp isn't the best solution to the case sensitivity problem, so we'll revisit it when we examine **string** comparisons.

Finding in reverse

Sometimes it's necessary to search through a **string** from end to beginning, if you need to find the data in "last in / first out " order. The string member function **rfind( )** handles this job.

```
//: C17:Rparse.cpp
// Reverse the order of words in a string
#include <string>
#include <iostream>
#include <vector>
using namespace std;
```

```
int main() {
  // The ';' characters will be delimiters
  string s("now.;sense;make;to;going;is;This");
  cout << s << endl;
  // To store the words:
  vector<string> strings;
  // The last element of the string:
  int last = s.size();
  // The beginning of the current word:
  int current = s.rfind(';');
  // Walk backward through the string:
  while(current != string::npos){
    // Push each word into the vector.
    // Current is incremented before copying to
    // avoid copying the delimiter.
    strings.push_back(
      s.substr(++current,last - current));
    // Back over the delimiter we just found,
    // and set last to the end of the next word
    current -= 2;
    last = current;
    // Find the next delimiter
    current = s.rfind(';', current);
  }
  // Pick up the first word - it's not
  // preceded by a delimiter
  strings.push_back(s.substr(0, last - current));
  // Print them in the new order:
  for(int j = 0; j < strings.size(); j++)
    cout << strings[j] << " ";

} ///:~
```

Here's how the output from **Rparse.cpp** looks:

```
now.;sense;make;to;going;is;This
This is going to make sense now.
```

rfind( ) backs through the string looking for tokens, reporting the array index of matching characters or **string::npos** if it is unsuccessful.

Finding first/last of a set

The **find_first_of( )** and **find_last_of( )** member functions can be conveniently put to work to create a little utility that will strip whitespace characters off of both ends of a string. Notice it doesn't touch the original string, but instead returns a new string:

```
//: C17:trim.h
#ifndef TRIM_H
#define TRIM_H
#include <string>
// General tool to strip spaces from both ends:
```

```
inline std::string trim(const std::string& s) {
  if(s.length() == 0)
    return s;
  int b = s.find_first_not_of(" \t");
  int e = s.find_last_not_of(" \t");
  if(b == -1) // No non-spaces
    return "";
  return std::string(s, b, e - b + 1);
}

#endif // TRIM_H ///:~
```

The first test checks for an empty **string**; in that case no tests are made and a copy is returned. Notice that once the end points are found, the **string** constructor is used to build a new **string** from the old one, giving the starting count and the length. This form also utilizes the “return value optimization” (see the index for more details).

Testing such a general-purpose tool needs to be thorough:

```
//: C17:TrimTest.cpp
#include "trim.h"
#include <iostream>
using namespace std;

string s[] = {
  " \t abcdefghijklmnop \t ",
  "abcdefghijklmnop \t ",
  " \t abcdefghijklmnop",
  "a", "ab", "abc", "a b c",
  " \t a b c \t ", " \t a \t b \t c \t ",
  "", // Must also test the empty string
};

void test(string s) {
  cout << "[" << trim(s) << "]" << endl;
}

int main() {
  for(int i = 0; i < sizeof s / sizeof *s; i++)
    test(s[i]);

} ///:~
```

In the array of **string s**, you can see that the character arrays are automatically converted to **string** objects. This array provides cases to check the removal of spaces and tabs from both ends, as well as ensuring that spaces and tabs do not get removed from the middle of a **string**.

Removing characters from strings

My word processor/page layout program (Microsoft Word) will save a document in HTML, but it doesn’t recognize that the code listings in this book should be tagged with the HTML “preformatted” tag (<PRE>), and it puts paragraph marks (<P> and </P>) around every listing line. This means that all the

indentation in the code listings is lost. In addition, Word saves HTML with reduced font sizes for body text, which makes it hard to read.

To convert the book to HTML form [49], then, the original output must be reprocessed, watching for the tags that mark the start and end of code listings, inserting the <PRE> and </PRE> tags at the appropriate places, removing all the <P> and </P> tags within the listings, and adjusting the font sizes. Removal is accomplished with the **erase( )** member function, but you must correctly determine the starting and ending points of the substring you wish to erase. Here's the program that reprocesses the generated HTML file:

```
//: C17:ReprocessHTML.cpp
// Take Word's html output and fix up
// the code listings and html tags
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// Produce a new string which is the original
// string with the html paragraph break marks
// stripped off:
string stripPBreaks(string s) {
  int br;
  while((br = s.find("<P>")) != string::npos)
    s.erase(br, strlen("<P>"));
  while((br = s.find("</P>")) != string::npos)
    s.erase(br, strlen("</P>"));
  return s;
}

// After the beginning of a code listing is
// detected, this function cleans up the listing
// until the end marker is found. The first line
// of the listing is passed in by the caller,
// which detects the start marker in the line.
void fixupCodeListing(istream& in,
  ostream& out, string& line, int tag) {
  out << line.substr(0, tag)
    << "<PRE>" // Means "preformatted" in html
    << stripPBreaks(line.substr(tag)) << endl;
  string s;
  while(getline(in, s)) {
    int endtag = s.find("/""/""/"":~");
    if(endtag != string::npos) {
      endtag += strlen("/""/""/"":~");
      string before = s.substr(0, endtag);
      string after = s.substr(endtag);
      out << stripPBreaks(before) << "</PRE>"
        << after << endl;
      return;
    }
    out << stripPBreaks(s) << endl;
```

```
  }
}

string removals[] = {
  "<FONT SIZE=2>",
  "<FONT SIZE=1>",
  "<FONT FACE=\"Times\" SIZE=1>",
  "<FONT FACE=\"Times\" SIZE=2>",
  "<FONT FACE=\"Courier\" SIZE=1>",
  "SIZE=1", // Eliminate all other '1' & '2' size
  "SIZE=2",
};
const int rmsz =
  sizeof(removals)/sizeof(*removals);

int main(int argc, char* argv[]) {
  requireArgs(argc, 2);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  ofstream out(argv[2]);
  string line;
  while(getline(in, line)) {
    // The "Body" tag only appears once:
    if(line.find("<BODY") != string::npos) {
      out << "<BODY BGCOLOR=\"#FFFFFF\" "
      "TEXT=\"#000000\">" << endl;
      continue; // Get next line
    }
    // Eliminate each of the removals strings:
    for(int i = 0; i < rmsz; i++) {
      int find = line.find(removals[i]);
      if(find != string::npos)
        line.erase(find, removals[i].size());
    }
    int tag1 = line.find("/""/"":");
    int tag2 = line.find("/""*"":");
    if(tag1 != string::npos)
      fixupCodeListing(in, out, line, tag1);
    else if(tag2 != string::npos)
      fixupCodeListing(in, out, line, tag2);
    else
      out << line << endl;
  }

} ///:~
```

Notice the lines that detect the start and end listing tags by indicating them with each character in quotes. These tags are treated in a special way by the logic in the **Extractcode.cpp** tool for extracting code listings. To present the code for the tool in the text of the book, the tag sequence itself must not occur in the listing. This was accomplished by taking advantage of a C++ preprocessor feature that causes text strings delimited by adjacent pairs of double quotes to be merged into a single string during the preprocessor pass of the build.

```
int tag1 = line.find("/""/"":");
```

The effect of the sequence of **char** arrays is to produce the starting tag for code listings.

Stripping HTML tags

Sometimes it's useful to take an HTML file and strip its tags so you have something approximating the text that would be displayed in the Web browser, only as an ASCII text file. The **string** class once again comes in handy. The following has some variation on the theme of the previous example:

```
//: C17:HTMLStripper.cpp
// Filter to remove html tags and markers
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

string replaceAll(string s, string f, string r) {
  unsigned int found = s.find(f);
  while(found != string::npos) {
    s.replace(found, f.length(), r);
    found = s.find(f);
  }
  return s;
}

string stripHTMLTags(string s) {
  while(true) {
    unsigned int left = s.find('<');
    unsigned int right = s.find('>');
    if(left==string::npos || right==string::npos)
      break;
    s = s.erase(left, right - left + 1);
  }
  s = replaceAll(s, "&lt;", "<");
  s = replaceAll(s, "&gt;", ">");
  s = replaceAll(s, "&amp;", "&");
  s = replaceAll(s, " ", " ");
  // Etc...
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  const int sz = 4096;
  char buf[sz];
  while(in.getline(buf, sz)) {
    string s(buf);
    cout << stripHTMLTags(s) << endl;
  }
```

```
} ///:~
```

The **string** class can replace one string with another but there's no facility for replacing all the strings of one type with another, so the **replaceAll( )** function does this for you, inside a **while** loop that keeps finding the next instance of the find string **f**. That function is used inside **stripHTMLTags** after it uses **erase( )** to remove everything that appears inside angle braces (' <' and ' >'). Note that I probably haven't gotten all the necessary replacement values, but you can see what to do (you might even put all the find-replace pairs in a table...). In **main( )** the arguments are checked, and the file is read and converted. It is sent to standard output so you must redirect it with ' >' if you want to write it to a file.

Comparing strings

Comparing strings is inherently different than comparing numbers. Numbers have constant, universally meaningful values. To evaluate the relationship between the magnitude of two strings, you must make a *lexical comparison* . Lexical comparison means that when you test a character to see if it is "greater than" or "less than" another character, you are actually comparing the numeric representation of those characters as specified in the collating sequence of the character set being used. Most often, this will be the ASCII collating sequence, which assigns the printable characters for the English language numbers in the range from 32 to 127 decimal. In the ASCII collating sequence, the first "character" in the list is the space, followed by several common punctuation marks, and then uppercase and lowercase letters. With respect to the alphabet, this means that the letters nearer the front have lower ASCII values than those nearer the end. With these details in mind, it becomes easier to remember that when a lexical comparison that reports s1 is "greater than" s2, it simply means that when the two were compared, the first differing character in s1 came later in the alphabet than the character in that same position in s2.

C++ provides several ways to compare strings, and each has their advantages. The simplest to use are the non member overloaded operator functions **operator ==, operator != operator >, operator <, operator >=,** and **operator <=.**

```
//: C17:CompStr.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  // Strings to compare
  string s1("This ");
  string s2("That ");
  for(int i = 0; i< s1.size() &&
    i < s2.size(); i++)
    // See if the string elements are the same:
    if(s1[i] == s2[i])
      cout << s1[i] << "  " << i << endl;
  // Use the string inequality operators
  if(s1 != s2) {
    cout << "Strings aren't the same:" << " ";
    if(s1 > s2)
      cout << "s1 is > s2" << endl;
    else
      cout << "s2 is > s1" << endl;
```

```
  }
} ///:~
```

Here's the output from **CompStr.cpp**:

```
T 0
h 1
  4
Strings aren't the same: s1 is > s2
```

The overloaded comparison operators are useful for comparing both full strings and individual string elements.

Notice in the code fragment below the flexibility of argument types on both the left and right hand side of the comparison operators. The overloaded operator set allows the direct comparison of string objects, quoted literals, and pointers to C style strings.

```
// The lvalue is a quoted literal and
// the rvalue is a string
if("That " == s2)
  cout << "A match" << endl;
// The lvalue is a string and the rvalue is a
// pointer to a c style null terminated string
if(s1 != s2.c_str())
cout << "No match" << endl;
```

You won't find the logical not (!) or the logical comparison operators (&& and ||) among operators for string. (Neither will you find overloaded versions of the bitwise C operators &, |, ^, or ~.) The overloaded non member comparison operators for the string class are limited to the subset which has clear, unambiguous application to single characters or groups of characters.

The **compare( )** member function offers you a great deal more sophisticated and precise comparison than the non member operator set, because it returns a lexical comparison value, and provides for comparisons that consider subsets of the string data. It provides overloaded versions that allow you to compare two complete strings, part of either string to a complete string, and subsets of two strings. This example compares complete strings:

```
//: C17:Compare.cpp
// Demonstrates compare(), swap()
#include <string>
#include <iostream>
using namespace std;

int main() {
  string first("This");
  string second("That");
  // Which is lexically greater?
  switch(first.compare(second)) {
    case 0: // The same
```

```
      cout << first << " and " << second <<
        " are lexically equal" << endl;
      break;
    case -1: // Less than
      first.swap(second);
      // Fall through this case...
    case 1: // Greater than
      cout << first <<
        " is lexically greater than " <<
        second << endl;
  }
} ///:~
```

The output from **Compare.cpp** looks like this:

```
This is lexically greater than That
```

To compare a subset of the characters in one or both strings, you add arguments that define where to start the comparison and how many characters to consider. For example, we can use the overloaded version of **compare( )**:

s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos, s2NumberChars);

If we substitute the above version of **compare( )** in the previous program so that it only looks at the first two characters of each string, the program becomes:

```
//: C17:Compare2.cpp
// Overloaded compare()
#include <string>
#include <iostream>
using namespace std;

int main() {
  string first("This");
  string second("That");
  // Compare first two characters of each string:
  switch(first.compare(0, 2, second, 0, 2)) {
    case 0: // The same
      cout << first << " and " << second <<
        " are lexically equal" << endl;
      break;
    case -1: // Less than
      first.swap(second);
      // Fall through this case...
    case 1: // Greater than
      cout << first <<
        " is lexically greater than " <<
        second << endl;
  }
} ///:~
```

The output is:

```
This and That are lexically equal
```

which is true, for the first two characters of "This" and "That."

Indexing with [ ] vs. at( )

In the examples so far, we have used C style array indexing syntax to refer to an individual character in a string. C++ strings provide an alternative to the **s[n]** notation: the **at( )** member. These two idioms produce the same result in C++ if all goes well:

```
//: C17:StringIndexing.cpp
#include <string>
#include <iostream>
using namespace std;
int main(){
  string s("1234");
  cout << s[1] << " ";
  cout << s.at(1) << endl;
} ///:~
```

The output from this code looks like this:

```
2 2
```

However, there is one important difference between **[ ]** and **at( )**. When you try to reference an array element that is out of bounds, **at( )** will do you the kindness of throwing an exception, while ordinary **[ ]** subscripting syntax will leave you to your own devices:

```
//: C17:BadStringIndexing.cpp
#include <string>
#include <iostream>
using namespace std;

int main(){
  string s("1234");
  // Runtime problem: goes beyond array bounds:
  cout << s[5] << endl;
  // Saves you by throwing an exception:
  cout << s.at(5) << endl;
} ///:~
```

Using **at( )** in place of **[ ]** will give you a chance to gracefully recover from references to array elements that don't exist. **at( )** throws an object of class **out_of_range.** By catching this object in an exception handler, you can take appropriate remedial actions such as recalculating the offending subscript or growing the array. (You can read more about Exception Handling in Chapter XX)

Using iterators

In the example program **NewFind.cpp**, we used a lot of messy and rather tedious C **char** array handling code to change the case of the characters in a string and then search for the occurrence of matches to a substring. Sometimes the "quick and dirty" method is justifiable, but in general, you won't want to sacrifice the advantages of having your string data safely and securely encapsulated in the C++ object where it lives.

Here is a better, safer way to handle case insensitive comparison of two C++ string objects. Because no data is copied out of the objects and into C style strings, you don't have to use pointers and you don't have to risk overwriting the bounds of an ordinary character array. In this example, we use the string **iterator**. Iterators are themselves objects which move through a collection or container of other objects, selecting them one at a time, but never providing direct access to the implementation of the container. Iterators are *not* pointers, but they are useful for many of the same jobs.

```
//: C17:CmpIter.cpp
// Find a group of characters in a string
#include <string>
#include <iostream>
using namespace std;

// Case insensitive compare function:
int
stringCmpi(const string& s1, const string& s2) {
  // Select the first element of each string:
  string::const_iterator
    p1 = s1.begin(), p2 = s2.begin();
  // Don't run past the end:
  while(p1 != s1.end() && p2 != s2.end()) {
    // Compare upper-cased chars:
    if(toupper(*p1) != toupper(*p2))
      // Report which was lexically  greater:
      return (toupper(*p1)<toupper(*p2))? -1 : 1;
    p1++;
    p2++;
  }
  // If they match up to the detected eos, say
  // which was longer. Return 0 if the same.
  return(s2.size() - s1.size());
}

int main() {
  string s1("Mozart");
  string s2("Modigliani");
  cout << stringCmpi(s1, s2) << endl;

} ///:~
```

Notice that the iterators **p1** and **p2** use the same syntax as C pointers – the ' ***' operator makes the *value of* element at the location given by the iterators available to the **toupper( )** function. **toupper( )** doesn't actually change the content of the element in the string. In fact, it can't. This definition of **p1** tells us that we can only use the elements **p1** points to as constants.

```
string::const_iterator p1 = s1.begin();
```

The way **toupper( )** and the iterators are used in this example is called a *case preserving* case insensitive comparison. This means that the string didn't have to be copied or rewritten to accommodate case insensitive comparison. Both of the strings retain their original data, unmodified.

Iterating in reverse

Just as the standard C pointer gives us the increment (++) and decrement (--) operators to make pointer arithmetic a bit more convenient, C++ string iterators come in two basic varieties. You've seen **end( )** and **begin( )**, which are the tools for moving forward through a string one element at a time. The reverse iterators **rend( )** and **rbegin( )** allow you to step backwards through a string. Here's how they work:

```
//: C17:RevStr.cpp
// Print a string in reverse
#include <string>
#include <iostream>
using namespace std;
int main() {
  string s("987654321");
  // Use this iterator to walk backwards:
  string::reverse_iterator rev;
  // "Incrementing" the reverse iterator moves
  // it to successively lower string elements:
  for(rev = s.rbegin(); rev != s.rend(); rev++)
    cout << *rev << " ";

} ///:~
```

The output from **RevStr.cpp** looks like this:

```
1 2 3 4 5 6 7 8 9
```

Reverse iterators act like pointers to elements of the string's character array, *except that when you apply the increment operator to them, they move backward rather than forward* . **rbegin( )** and **rend( )** supply string locations that are consistent with this behavior, to wit, **rbegin( )** locates the position just beyond the end of the string, and **rend( )** locates the beginning. Aside from this, the main thing to remember about reverse iterators is that they *aren't* type equivalent to ordinary iterators. For example, if a member function parameter list includes an iterator as an argument, you can't substitute a reverse iterator to get the function to perform it's job walking backward through the string. Here's an illustration:

```
// The compiler won't accept this

string sBackwards(s.rbegin(), s.rend());
```

The string constructor won't accept reverse iterators in place of forward iterators in its parameter list. This is also true of string members such as **copy( )**, **insert( ),** and **assign( ).**

Strings and character traits

We seem to have worked our way around the margins of case insensitive string comparisons using C++

string objects, so maybe it's time to ask the obvious question: "Why isn't case-insensitive comparison part of the standard **string** class ?" The answer provides interesting background on the true nature of C++ string objects.

Consider what it means for a character to have "case." Written Hebrew, Farsi, and Kanji don't use the concept of upper and lower case, so for those languages this idea has no meaning at all. This the first impediment to built-in C++ support for case-insensitive character search and comparison: the idea of case sensitivity is not universal, and therefore not portable.

It would seem that if there were a way of designating that some languages were "all uppercase" or "all lowercase" we could design a generalized solution. However, some languages which employ the concept of "case" *also* change the meaning of particular characters with diacritical marks: the cedilla in Spanish, the circumflex in French, and the umlaut in German. For this reason, any case-sensitive collating scheme that attempts to be comprehensive will be nightmarishly complex to use.

Although we usually treat the C++ **string** as a class, this is really not the case. **string** is a **typedef** of a more general constituent, the **basic_string< >** template. Observe how **string** is declared in the standard C++ header file:

```
typedef basic_string<char> string;
```

To really understand the nature of strings, it's helpful to delve a bit deeper and look at the template on which it is based. Here's the declaration of the **basic_string< >** template:

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >

class basic_string;
```

Earlier in this book, templates were examined in a great deal of detail. The main thing to notice about the two declarations above are that the **string** type is created when the **basic_string** template is instantiated with **char.** Inside the **basic_string< >** template declaration, the line

```
class traits = char_traits<charT>,
```

tells us that the behavior of the class made from the **basic_string< >** template is specified by a class based on the template **char_traits< >**. Thus, the **basic_string< >** template provides for cases where you need string oriented classes that manipulate types other than **char** (wide characters or unicode, for example). To do this, the **char_traits< >** template controls the content and collating behaviors of a variety of character sets using the character comparison functions **eq( )** (equal), **ne( )** (not equal), and **lt( )** (less than) upon which the **basic_string< >** string comparison functions rely.

This is why the string class doesn't include case insensitive member functions: That's not in its job description. To change the way the string class treats character comparison, you must supply a different of **char_traits< >** template, because that defines the behavior of the individual character comparison member functions.

This information can be used to make a new type of **string** class that ignores case. First, we'll define a

new case insensitive **char_traits< >** template that inherits the existing one. Next, we'll override only the members we need to change in order to make character-by-character comparison case insensitive. (In addition to the three lexical character comparison members mentioned above, we'll also have to supply new implementation of **find( )** and **compare( )**.) Finally, we'll **typedef** a new class based on **basic_string**, but using the case insensitive **ichar_traits** template for its second argument.

```
//: C17:ichar_traits.h
// Creating your own character traits
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <string>
#include <cctype>

struct ichar_traits : std::char_traits<char> {
  // We'll only change character by
  // character comparison functions
  static bool eq(char c1st, char c2nd) {
    return
      std::toupper(c1st) == std::toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return
      std::toupper(c1st) != std::toupper(c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return
      std::toupper(c1st) < std::toupper(c2nd);
  }
  static int compare(const char* str1,
    const char* str2, size_t n) {
    for(int i = 0; i < n; i++) {
      if(std::tolower(*str1)>std::tolower(*str2))
        return 1;
      if(std::tolower(*str1)<std::tolower(*str2))
        return -1;
      if(*str1 == 0 || *str2 == 0)
        return 0;
      str1++; str2++; // Compare the other chars
    }
    return 0;
  }
  static const char* find(const char* s1,
    int  n, char c) {
    while(n-- > 0 &&
      std::toupper(*s1) != std::toupper(c))
      s1++;
    return s1;
  }
};

#endif // ICHAR_TRAITS_H ///:~
```

If we **typedef** an **istring** class like this:

```
typedef basic_string<char, ichar_traits,
allocator<char> > istring;
```

Then this **istring** will act like an ordinary **string** in every way, except that it will make all comparisons without respect to case. Here's an example:

```
//: C17:ICompare.cpp
#include "ichar_traits.h"
#include <string>
#include <iostream>
using namespace std;

typedef basic_string<char, ichar_traits,
  allocator<char> > istring;

int main() {
  // The same letters except for case:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first.compare(second) << endl;
} ///:~
```

The output from the program is "0", indicating that the strings compare as equal. This is just a simple example – in order to make **istring** fully equivalent to **string**, we'd have to create the other functions necessary to support the new **istring** type.

---

[49] I subsequently found better tools to accomplish this task, but the program is still interesting.

Contents | Prev | Next