13.2. Description of the String Classes

13.2.1. String Types

Header File

All types and functions for strings are defined in the header file `<string>`:

```
#include <string>
```

As usual, it defines all identifiers in namespace `std`.

Class Template `basic_string<>`

Inside `<string>`, class `basic_string<>` is defined as a basic type for all string types:

Click here to view code image

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT>,
              typename Allocator = allocator<charT> >
            class  basic_string;
}
```

This class is parametrized by the character type, the traits of the character type, and the memory model:

- The first parameter is the data type of a single character.
- The optional second parameter is a traits class, which provides all core operations for the characters of the string class. Such a traits class specifies how to copy or to compare characters (see Section 16.1.4, page 853, for details). If it is not specified, the default traits class according to the current character type is used. See Section 13.2.15, page 689, for a user-defined traits class that lets strings behave in a case-insensitive manner.
- The third optional argument defines the memory model that is used by the string class. As usual, the default value is the default memory model `allocator` (see Section 4.6, page 57, and Chapter 19 for details).

Concrete String Types

The C++ standard library provides a couple of specializations of class `basic_string<>`:

- Class `string` is the predefined specialization of that template for characters of type `char`:

```
namespace std {
    typedef basic_string<char> string;
}
```

- For strings that use wider character sets, such as Unicode or some Asian character sets, three other types are predefined (`u16string` and `u32string` are provided since C++11):

Click here to view code image

```
namespace std {
    typedef basic_string<wchar_t>  wstring;
    typedef basic_string<char16_t> u16string;
    typedef basic_string<char32_t> u32string;
}
```

See Chapter 16 for details about internationalization.

In the following sections, no distinction is made between these types of strings. The usage and the problems are the same because all string classes have the same interface. So, "string" means any string type: `string`, `wstring`, `u16string`, and `u32string`. The examples in this book usually use type `string` because the European and Anglo-American environments are the common environments for software development.

13.2.2. Operation Overview

Table 13.1 (see previous page) lists all operations that are provided for strings.

Table 13.1. String Operations

Operation	Effect
constructors	Create or copy a string
destructor	Destroys a string
=, assign()	Assign a new value
swap()	Swaps values between two strings
+=, append(), push_back()	Append characters
insert()	Inserts characters
erase(), pop_back()	Deletes characters (pop_back() since C++11)
clear()	Removes all characters (empties a string)
resize()	Changes the number of characters (deletes or appends characters at the end)
replace()	Replaces characters
+	Concatenates strings
==, !=, <, <=, >, >=, compare()	Compare strings
empty()	Returns whether the string is empty
size(), length()	Return the number of characters
max_size()	Returns the maximum possible number of characters
capacity()	Returns the number of characters that can be held without reallocation
reserve()	Reserves memory for a certain number of characters
shrink_to_fit()	Shrinks the memory for the current number of characters (since C++11)
[], at()	Access a character
front(), back()	Access the first or last character (since C++11)
>>, getline()	Read the value from a stream
<<	Writes the value to a stream
stoi(), stol(), stoll()	Convert string to signed integral value (since C++11)
stoul(), stoull()	Convert string to unsigned integral value (since C++11)
stof(), stod(), stold()	Convert string to floating-point value (since C++11)
to_string(), to_wstring()	Convert integral/floating-point value to string (since C++11)
copy()	Copies or writes the contents to a character array
data(), c_str()	Returns the value as C-string or character array
substr()	Returns a certain substring
find functions	Search for a certain substring or character
begin(), end()	Provide normal iterator support
cbegin(), cend()	Provide constant iterator support (since C++11)
rbegin(), rend()	Provide reverse iterator support
crbegin(), crend()	Provide constant reverse iterator support (since C++11)
get_allocator()	Returns the allocator

String Operation Arguments

Many operations are provided to manipulate strings. In particular, the operations that manipulate the value of a string have several overloaded versions that specify the new value with one, two, or three arguments. All these operations use the argument scheme of Table 13.2.

Table 13.2. Scheme of String Operation Arguments

Arguments	Interpretation
`const string& str`	The whole string *str*
`const string& str, size_type idx, size_type num`	At most, the first *num* characters of *str* starting with index *idx*
`const char* cstr`	The whole C-string *cstr*
`const char* chars, size_type len`	*len* characters of the character array *chars*
`char c`	The character *c*
`size_type num, char c`	*num* occurrences of character *c*
`const_iterator beg, const_iterator end`	All characters in range [*beg*,*end*)
initlist	All characters in *initlist* (since C++11)

Note that only the single-argument version `const char*` handles the character `'\0'` as a special character that terminates the string. In all other cases, `'\0'` is *not* a special character:

Click here to view code image

```
std::string s1("nico");        // initializes s1 with: 'n' 'i' 'c' 'o'
std::string s2("nico",5);      // initializes s2 with: 'n' 'i' 'c' 'o' '\0'
std::string s3(5,'\0');        // initializes s3 with: '\0' '\0' '\0' '\0' '\0'

s1.length()                    // yields 4
s2.length()                    // yields 5
s3.length()                    // yields 5
```

Thus, in general a string might contain any character. In particular, a string might contain the contents of a binary file.

Passing a null pointer as *cstr* results in undefined behavior.

See Table 13.3 for an overview of which operation uses which kind of arguments. All operators can handle only objects as single values. Therefore, to assign, compare, or append a part of a string or C-string, you must use the function that has the corresponding name.

Table 13.3. Available Operations Having String Parameters

	Full String	Part of String	C-string (*char**)	*char* Array	Single *char*	*num chars*	Iterator Range	Init list
constructors	Yes	Yes	Yes	Yes	—	Yes	Yes	Yes
=	Yes	—	Yes	—	Yes	—	—	Yes
`assign()`	Yes	Yes	Yes	Yes	—	Yes	Yes	Yes
+=	Yes	—	Yes	—	Yes	—	—	Yes
`append()`	Yes	Yes	Yes	Yes	—	Yes	Yes	Yes
`push_back()`	—	—	—	—	Yes	—	—	—
`insert()` for idx	Yes	Yes	Yes	Yes	—	Yes	—	—
`insert()` for iter.	—	—	—	—	Yes	Yes	Yes	Yes
`replace()` for idx	Yes	Yes	Yes	Yes	Yes	Yes	—	—
`replace()` for iter.	Yes	—	Yes	Yes	—	Yes	Yes	Yes
find functions	Yes	—	Yes	Yes	Yes	—	—	—
+	Yes	—	Yes	—	Yes	—	—	—
==, !=, <, <=, >, >=	Yes	—	Yes	—	—	—	—	—
`compare()`	Yes	Yes	Yes	Yes	—	—	—	—

Operations Not Provided

The string classes of the C++ standard library do not solve every possible string problem. In fact, they do not provide direct solutions for regular expressions and text processing. Regular expressions, however, are covered by a separate library introduced with C++11 (see Chapter 14). For text processing (capitalization, case-insensitive comparisons), see Section 13.2.14, page 684, for some examples.

13.2.3. Constructors and Destructor

Table 13.4 lists all the constructors and the destructor for strings.

You can't initialize a string with a single character. Instead, you must use its address or an additional number of occurrences or the format of an initializer list (since C++11):

Click here to view code image

```
std::string s('x');     // ERROR
std::string s(1,'x');   // OK, creates a string that has one character 'x'
std::string s({'x'});   // OK, ditto (since C++11)
```

Table 13.4. Constructors and Destructor of Strings

Expression	Effect
`string s`	Creates the empty string *s*
`string s(`*str*`)`	Copy constructor; creates a string as a copy of the existing string *str*
`string s(`*rvStr*`)`	Move constructor; creates a string and moves the contents of *rvStr* to it (*rvStr* has a valid state with undefined value afterward)
`string s(`*str*`,`*stridx*`)`	Creates a string *s* that is initialized by the characters of string *str* starting with index *stridx*
`string s(`*str*`,`*stridx*`,`*strlen*`)`	Creates a string *s* that is initialized by, at most, *strlen* characters of string *str* starting with index *stridx*
`string s(`*cstr*`)`	Creates a string *s* that is initialized by the C-string *cstr*
`string s(`*chars*`,`*charslen*`)`	Creates a string *s* that is initialized by *charslen* characters of the character array *chars*
`string s(`*num*`,`*c*`)`	Creates a string that has *num* occurrences of character *c*
`string s(`*beg*`,`*end*`)`	Creates a string that is initialized by all characters of the range [*beg*,*end*)
`string s(`*initlist*`)`	Creates a string that is initialized by all characters in *initlist* (since C++11)
`s.~string()`	Destroys all characters and frees the memory

This means that there is an automatic type conversion from type `const char*` but not from type `char` to type `string`.

The initialization by a range that is specified by iterators is described in Section 13.2.14, page 684.

13.2.4. Strings and C-Strings

In standard C++, the type of string literals was changed from `char*` to `const char*`. However, to provide backward compatibility, there is an implicit but deprecated conversion to `char*` for them. Because string literals don't have type `string`, there is a strong relationship between `string` class objects and ordinary C-strings: You can use ordinary C-strings in almost every situation where strings are combined with other string-like objects (comparing, appending, inserting, etc.). In particular, there is an automatic type conversion from `const char*` into strings. However, there is *no* automatic type conversion from a string object to a C-string. This is for safety reasons. It prevents unintended type conversions that result in strange behavior (type `char*` often has strange behavior) and ambiguities. For example, in an expression that combines a `string` and a C-string, it would be possible to convert `string` into `char*` and vice versa. Instead, there are several ways to create or write/copy in a C-string. In particular, `c_str()` is provided to generate the value of a string as a C-string as a character array that has `'\0'` as its last character. By using `copy()`, you can copy or write the value to an existing C-string or character array.

Note that strings do *not* provide a special meaning for the character `'\0'`, which is used as a special character in an ordinary C-string to mark the end of the string. The character `'\0'` may be part of a string just like every other character.

Note also that if you use an old-style null pointer (`NULL`) instead of `nullptr` (see Section 3.1.1, page 14) or a `char*` parameter, strange behavior results. The reason is that `NULL` has an integral type and is interpreted as the number `0` or the character with value `0` if the operation is overloaded for a single integral type. So you should always use `nullptr` or `char*` pointers.

There are three possible ways to convert the contents of the string into a raw array of characters or C-string:

1. **`data()`** and **`c_str()`** return the contents of the string as an array of characters. The array includes the *end-of-string* character at position `[size()]`, so for `string`s, the result is a valid C-string including `'\0'`.

   Note that before C++11, the return type of `data()` was *not* a valid C-string, because no `'\0'` character was guaranteed to get appended.

2. **copy()** copies the contents of the string into a character array provided by the caller. An `'\0'` character is not appended.

Note that `data()` and `c_str()` return an array that is owned by the string. Thus, the caller must not modify or free the memory. For example:

[Click here to view code image]

```
std::string s("12345");

atoi(s.c_str())            // convert string into integer
f(s.data(),s.length())     // call function for a character array
                           // and the number of characters

char buffer[100];
s.copy(buffer,100);        // copy at most 100 characters of s into buffer
s.copy(buffer,100,2);      // copy at most 100 characters of s into buffer
                           // starting with the third character of s
```

You usually should use strings in the whole program and convert them into C-strings or character arrays only immediately before you need the contents as type `char*`. Note that the return value of `c_str()` and `data()` is valid only until the next call of a nonconstant member function for the same string:

[Click here to view code image]

```
std::string s;
...
foo(s.c_str());     // s.c_str() is valid during the whole statement

const char* p;
p = s.c_str();      // p refers to the contents of s as a C-string
foo(p);             // OK (p is still valid)
s += "ext";         // invalidates p
foo(p);             // ERROR: argument p is not valid
```

13.2.5. Size and Capacity

To use strings effectively and correctly, you need to understand how the size and capacity of strings cooperate. For strings, three "sizes" exist:

1. **size()** and **length()** are equivalent functions that return the current number of characters of the string.[5]

[5] In this case, two member functions do the same thing because `length()` returns the length of the string, as `strlen()` does for ordinary C-strings, whereas `size()` is the common member function for the number of elements according to the concept of the STL.

   The **empty()** member function is a shortcut for checking whether the number of characters is zero. Thus, it checks whether the string is empty. Because it might be faster, you should use `empty()` instead of `length()` or `size()`.

2. **max_size()** returns the maximum number of characters a string may contain. A string typically contains all characters in a single block of memory, so there might be relevant restrictions on PCs. Otherwise, this value usually is the maximum value of the type of the index less one. It is "less one" for two reasons: (a) The maximum value itself is `npos`, and (b) an implementation might append `'\0'` internally at the end of the internal buffer so that it simply returns that buffer when the string is used as a C-string (for example, by `c_str()`). Whenever an operation results in a string that has a length greater than `max_size()`, the class throws `length_error`.

3. **capacity()** returns the number of characters a string could contain without having to reallocate its internal memory.

Having sufficient capacity is important for two reasons:

1. Reallocation invalidates all references, pointers, and iterators that refer to characters of the string.
2. Reallocation takes time.

Thus, the capacity must be taken into account if a program uses pointers, references, or iterators that refer to a string or to characters of a string, or if speed is a goal.

The member function `reserve()` is provided to avoid reallocations. `reserve()` lets you reserve a certain capacity before you really need it to ensure that references are valid as long as the capacity is not exceeded:

```
std::string s;      // create empty string
s.reserve(80);      // reserve memory for 80 characters
```

The concept of capacity for strings is, in principle, the same as for vector containers (see Section 7.3.1, page 270). However, there is one big difference: Unlike with vectors, calling `reserve()` for strings might be a call to shrink the capacity. Calling `reserve()` with an argument that is less than the current capacity is, in effect, a nonbinding shrink request. If the argument is less than the current number of characters, it is a nonbinding shrink-to-fit request. Thus, although you might *want* to shrink the capacity, it is not guaranteed to happen. The default value of `reserve()` for string is `0`. So, a call of `reserve()` without any argument is always a nonbinding shrink-to-fit request:

```
s.reserve();         // "would like to shrink capacity to fit the current size"
```

Since C++11, `shrink_to_fit()` provides the same effect:

```
s.shrink_to_fit(); // "would like to shrink capacity to fit the current size" (C++11)
```

A call to shrink capacity is nonbinding because how to reach an optimal performance is implementation-defined. Implementations of the string class might have different design approaches with respect to speed and memory usage. Therefore, implementations might increase capacity in larger steps and might never shrink the capacity.

The standard, however, specifies that capacity may shrink only because of a call of `reserve()` or `shrink_to_fit()` . Thus, it is guaranteed that references, pointers, and iterators remain valid even when characters are deleted or modified, provided that they refer to characters having a position that is before the manipulated characters.

13.2.6. Element Access

A string allows you to have read or write access to the characters it contains. You can access a single character via the subscript operator `[]` and the `at()` member function. Since C++11, `front()` and `back()` are provided to also access the first or last character, respectively.

All these operations return a reference to the character at the position of the passed index, which is a constant character if the string is constant. As usual, the first character has index `0` , and the last character has index `length()-1` . However, note the following differences:

- Operator `[]` does *not* check whether the index passed as an argument is valid; `at()` does. If called with an invalid index, `at()` throws an `out_of_range` exception. If operator `[]` is called with an invalid index, the behavior is undefined. The effect might be an illegal memory access that might then cause some nasty side effects or a crash (you're lucky if the result is a crash, because then you know that you did something wrong).
- In general, the position after the last character is valid. Thus, the current number of characters is a valid index. The operator returns the value that is generated by the default constructor of the character type. Thus, for objects of type `string` it returns the `char '\0'` .[6]

[6] Before C++11, for the nonconstant version of operator `[]` , the current number of characters was an invalid index. Using it did result in undefined behavior.

- `front()` is equivalent to `[0]` , which means that for empty strings the character representing the end of the string (`'\0'` for `string` s) is returned.
- For `at()` , the current number of characters is not a valid index.
- When called for an empty string, `back()` results in undefined behavior.

For example:

Click here to view code image

```
const std::string cs("nico");   // cs contains: 'n' 'i' 'c' 'o'
std::string s("abcde");         // s contains: 'a' 'b' 'c' 'd' 'e'
std::string t;                  // t contains no character (is empty)

s[2]                            // yields 'c' as char&
s.at(2)                         // yields 'c' as char&
s.front()                       // yields 'a' as char&
cs[2]                           // yields 'c' as const char&
cs.back()                       // yields 'o' as const char&

s[100]                          // ERROR: undefined behavior
s.at(100)                       // throws out_of_range
t.front()                       // yields '\0'
t.back()                        // ERROR: undefined behavior

s[s.length()]                   // yields '\0' (undefined behavior before
C++11)
cs[cs.length()]                 // yields '\0'
s.at(s.length())                // throws out_of_range
cs.at(cs.length())              // throws out_of_range
```

To enable you to modify a character of a string, the nonconstant versions of `[]` , `at()` , `front()` , and `back()` return a character reference. Note that this reference becomes invalid on reallocation:

Click here to view code image

```
std::string s("abcde");    // s contains: 'a' 'b' 'c' 'd' 'e'

char& r = s[2];           // reference to third character
```

```
char* p = &s[3];            // pointer to fourth character

r = 'X';                    // OK, s contains: 'a' 'b' 'X' 'd' 'e'
*p = 'Y';                   // OK, s contains: 'a' 'b' 'X' 'Y' 'e'

s = "new long value";       // reallocation invalidates r and p

r = 'X';                    // ERROR: undefined behavior
*p = 'Y';                   // ERROR: undefined behavior
```

Here, to avoid runtime errors, you would have had to `reserve()` enough capacity before `r` and `p` were initialized.

References, pointers, and iterators that refer to characters of a string may be invalidated by the following operations:[7]

[7] Before C++11, `data()` and `c_str()` also could invalidate references, iterators, and pointers to strings.

- If the value is swapped with `swap()`
- If a new value is read by `operator>>()` or `getline()`
- If any nonconstant member function is called, except operator `[]`, `at()`, `begin()`, `end()`, `rbegin()`, and `rend()`

See Section 13.2.14, page 684, for details about string iterators.

13.2.7. Comparisons

The usual comparison operators are provided for strings. The operands may be strings or C-strings:

Click here to view code image

```
std::string s1, s2;
...
s1 == s2        // returns true if s1 and s2 contain the same characters
s1 < "hello"    // return whether s1 is less than the C-string "hello"
```

If strings are compared by `<`, `<=`, `>`, or `>=`, their characters are compared lexicographically according to the current character traits. For example, all of the following comparisons yield `true`:

```
std::string("aaaa") < std::string("bbbb")
std::string("aaaa") < std::string("abba")
std::string("aaaa") < std::string("aaaaaa")
```

By using the `compare()` member functions, you can compare substrings. The `compare()` member functions can process more than one argument for each string, so you can specify a substring by its index and its length. Note that `compare()` returns an integral value rather than a Boolean value. This return value has the following meaning: `0` means equal, a value less than `0` means less than, and a value greater than `0` means greater than. For example:

Click here to view code image

```
std::string s("abcd");

s.compare("abcd")           // returns 0
s.compare("dcba")           // returns a value < 0 (s is less)
s.compare("ab")             // returns a value > 0 (s is greater)

s.compare(s)                // returns 0 (s is equal to s)
s.compare(0,2,s,2,2)        // returns a value < 0 ("ab" is less than "cd")
s.compare(1,2,"bcx",2)      // returns 0 ("bc" is equal to "bc")
```

To use a different comparison criterion, you can define your own comparison criterion and use STL comparison algorithms (see Section 13.2.14, page 684, for an example), or you can use special character traits that make comparisons on a case-insensitive basis. However, because a string type that has a special traits class is a different data type, you cannot combine or process these strings with objects of type `string`. See Section 13.2.15, page 689, for an example.

In programs for the international market, it might be necessary to compare strings according to a specific locale. Class `locale` provides the parenthesis operator as a convenient way to do this (see Section 16.3, page 868). It uses the string collation facet, which is provided to compare strings for sorting according to some locale conventions. See Section 16.4.5, page 904, for details.

13.2.8. Modifiers

You can modify strings by using different member functions and operators.

Assignments

To modify a string, you can use operator `=` to assign a new value. The assigned value may be a string, a C-string, or a single character. In addition, you can use the `assign()` member functions to assign strings when more than one argument is needed to describe the new value. For example:

Click here to view code image

```
const std::string aString("othello");
std::string s;

s = aString;                // assign "othello"
s = "two\nlines";           // assign a C-string
s = ' ';                    // assign a single character

s.assign(aString);          // assign "othello" (equivalent to operator =)
s.assign(aString,1,3);      // assign "the"
s.assign(aString,2,std::string::npos);      // assign "hello"

s.assign("two\nlines");     // assign a C-string (equivalent to operator =)
s.assign("nico",5);         // assign the character array: 'n' 'i' 'c' 'o' '\0'
s.assign(5,'x');            // assign five characters: 'x' 'x' 'x' 'x' 'x'
```

You also can assign a range of characters that is defined by two iterators. See Section 13.2.14, page 684, for details.

Swapping Values

As with many nontrivial types, the string type provides a specialization of the `swap()` function, which swaps the contents of two strings (the global `swap()` function was introduced in Section 5.5.2, page 136). The specialization of `swap()` for strings guarantees constant complexity, so you should use it to swap the value of strings and to assign strings if you don't need the assigned string after the assignment.

Making Strings Empty

To remove all characters in a string, you have several possibilities. For example:

```
std::string s;

s = "";         // assign the empty string
s.clear();      // clear contents
s.erase();      // erase all characters
```

Inserting and Removing Characters

There are numerous member functions to insert, remove, replace, and erase characters of a string. To append characters, you can use operator `+=`, `append()`, and `push_back()`. For example:

Click here to view code image

```
const std::string aString("othello");
std::string s;

s += aString;               // append "othello"
s += "two\nlines";          // append C-string
s += '\n';                  // append single character
s += { 'o', 'k' };          // append an initializer list of characters (since C++11)

s.append(aString);          // append "othello" (equivalent to operator +=)
s.append(aString,1,3);      // append "the"
s.append(aString,2,std::string::npos);      // append "hello"

s.append("two\nlines");     // append C-string (equivalent to operator +=)
s.append("nico",5);         // append character array: 'n' 'i' 'c' 'o' '\0'
s.append(5,'x');            // append five characters: 'x' 'x' 'x' 'x' 'x'

s.push_back('\n');          // append single character (equivalent to operator +=)
```

Operator `+=` appends single-argument values, including initializer lists of characters since C++11. `append()` is overloaded for different arguments. One version of `append()` lets you append a range of characters specified by two iterators (see Section 13.2.14, page 684). The `push_back()` member function is provided for back inserters so that STL algorithms are able to append characters to a string (see Section 9.4.2, page 455, for details about back inserters and Section 13.2.14, page 688, for an example of their use with strings).

Similar to `append()`, several `insert()` member functions enable you to insert characters. These functions require the index of the character, after which the new characters are inserted:

```
const std::string aString("age");
```

```
std::string s("p");

s.insert(1,aString);       // s: page
s.insert(1,"ersifl");      // s: persiflage
```

Note that no `insert()` member function is provided to pass the index and a single character. Thus, you must pass a string or an additional number:

```
s.insert(0,' ');     // ERROR
s.insert(0," ");     // OK
```

You might also try

```
s.insert(0,1,' ');    // ERROR: ambiguous
```

However, this results in a nasty ambiguity because `insert()` is overloaded for the following signatures:

```
insert (size_type  idx, size_type num, charT c);   // position is index
insert (iterator   pos, size_type num, charT c);   // position is iterator
```

For type `string` , `size_type` is usually defined as `unsigned` , and `iterator` is often defined as `char*` . In this case, the first argument `0` has two equivalent conversions. So, to get the correct behavior, you have to write:

```
s.insert((std::string::size_type)0,1,' ');   // OK
```

The second interpretation of the ambiguity described here is an example of the use of iterators to insert characters. If you wish to specify the insert position as an iterator, you can do it in three ways: insert a single character, insert a certain number of the same character, and insert a range of characters specified by two iterators (see Section 13.2.14, page 684).

Similar to `append()` and `insert()` , several `erase()` functions and `pop_back()` (since C++11) remove characters, and several `replace()` functions replace characters. For example:

Click here to view code image

```
std::string s = "i18n";                    // s: i18n
s.replace(1,2,"nternationalizatio");       // s: internationalization
s.erase(13);                               // s: international
s.erase(7,5);                              // s: internal
s.pop_back();                              // s: interna (since C++11)
s.replace(0,2,"ex");                       // s: externa
```

You can use `resize()` to change the number of characters. If the new size that is passed as an argument is less than the current number of characters, characters are removed from the end. If the new size is greater than the current number of characters, characters are appended at the end. You can pass the character that is appended if the size of the string grows. If you don't, the default constructor for the character type is used (which is the `'\0'` character for type `char`).

13.2.9. Substrings and String Concatenation

You can extract a substring from any string by using the `substr()` member function. For example:

```
std::string s("interchangeability");

s.substr()               // returns a copy of s
s.substr(11)             // returns string("ability")
s.substr(5,6)            // returns string("change")
s.substr(s.find('c'))    // returns string("changeability")
```

You can use operator `+` to concatenate two strings or C-strings or one of those with single characters For example, the statements

```
std::string s1("enter");
std::string s2("nation");
std::string i18n;

i18n = 'i' + s1.substr(1) + s2 + "aliz" + s2.substr(1);
std::cout << "i18n means: " + i18n << std::endl;
```

have the following output:

```
i18n means: internationalization
```

Since C++11, operator `+` is also overloaded for strings that are rvalue references to support the move semantics. Thus, if a string argument passed to operator `+` is no longer needed afterward, you should use `move()` to pass it to the operator. For example:

Click here to view code image

```
string foo()
{
    std::string s1("international");
    std::string s2("ization");

     std::string s = std::move(s1) + std::move(s2);      // OK
                     // s1 and s2 have valid state with unspecified value
     return s;
}
```

13.2.10. Input/Output Operators

The usual I/O operators are defined for strings:

- **Operator** >> reads a string from an input stream.
- **Operator** << writes a string to an output stream.

These operators behave as they do for ordinary C-strings. In particular, operator `>>` operates as follows:

1. It skips leading whitespaces if the `skipws` flag (see Section 15.7.7, page 789) is set.
2. It reads all characters until any of the following happens:
   – The next character is a whitespace.
   – The stream is no longer in a good state (for example, due to end-of-file).
   – The current `width()` of the stream (see Section 15.7.3, page 781) is greater than `0`, and `width()` characters are read.
   – `max_size()` characters are read.
3. It sets `width()` of the stream to `0`.

Thus, in general, the input operator reads the next word while skipping leading whitespaces. A whitespace is any character for which `isspace(c,` *strm* `.getloc())` is `true` (`isspace()` is explained in Section 16.4.4, page 895).

The output operator also takes the `width()` of the stream into consideration. That is, if `width()` is greater than `0`, operator `<<` writes at least `width()` characters.

Note also that since C++11, operators `<<` and `>>` are declared to process rvalue references to streams. This, for example, allows you to use temporary string streams (see Section 15.10.2, page 806, for details).

getline()

The string classes also provide a special convenience function `std::getline()` for reading line by line: This function reads all characters, including leading whitespaces, until the line delimiter or end-of-file is reached. The line delimiter is extracted but not appended. By default, the line delimiter is the newline character, but you can pass your own "line" delimiter as an optional argument.[8] This way, you can read token by token, separated by any arbitrary character:

[8] You don't have to qualify `getline()` with `std::` because when calling a function *argument dependent lookup* (ADL, also known as *Koenig lookup*) will always consider the namespace where the class of an argument was defined.

Click here to view code image

```
std::string s;

while (getline(std::cin,s)) {          // for each line read from cin
    …
}

while (getline(std::cin,s,':')) {      // for each token separated by ':'
    …
}
```

Note that if you read token by token, the newline character is not a special character. In this case, the tokens might contain a newline character.

Note also that since C++11, `getline()` is overloaded for both lvalue and rvalue stream references, which allows using temporary string streams:

Click here to view code image

```
void process (const std::string& filecontents)
{
    // process first line of passed string:
    std::string firstLine;
```

```
        std::getline(std::stringstream(filecontents),   // OK since C++11
                     firstLine);
        ...
    }
```

See Section 15.10, page 802, for details about string streams.

13.2.11. Searching and Finding

The C++ standard library provides many abilities to search and find characters or substrings in a string:[9]

[9] Don't be confused because I write about searching "and" finding. They are almost synonymous. The search functions use "find" in their names. However, unfortunately, they don't guarantee to find anything. In fact, they "search" for something or "try to find" something. So I use the term *search* for the behavior of these functions and *find* with respect to their names.

- By using **member functions**, you can search
  - A single character, a character sequence (substring), or one of a certain set of characters
  - Forward and backward
  - Starting from any position at the beginning or inside the string
- By using the **regex library** (see Chapter 14), you can search for more complicated patterns of character sequences. See Section 13.2.14, page 687, for an example.
- By using **STL algorithms**, you can also search for single characters or specific character sequences (see Section 11.2.2, page 507). Note that these algorithms allow you to use your own comparison criterion (see Section 13.2.14, page 684, for an example).

Member Functions for Searching and Finding

All search functions have the word *find* inside their name. They try to find a character position given a *value* that is passed as an argument. How the search proceeds depends on the exact name of the find function. Table 13.5 lists all the search functions for strings.

Table 13.5. Search Functions for Strings

String Function	Effect
`find()`	Finds the first occurrence of *value*
`rfind()`	Finds the last occurrence of *value* (reverse find)
`find_first_of()`	Finds the first character that is part of *value*
`find_last_of()`	Finds the last character that is part of *value*
`find_first_not_of()`	Finds the first character that is not part of *value*
`find_last_not_of()`	Finds the last character that is not part of *value*

All search functions return the index of the first character of the character sequence that matches the search. If the search fails, they return `npos`. The search functions use the following argument scheme:

- The first argument is always the value that is searched for.
- The second optional value indicates an index at which to start the search in the string.
- The optional third argument is the number of characters of the value to search.

Unfortunately, this argument scheme differs from that of the other string functions. With the other string functions, the starting index is the first argument, and the value and its length are adjacent arguments. In particular, each search function is overloaded with the following set of arguments:

- `const` ***string*** `&` *value*

  searches against the characters of the string *value*.
- `const` ***string*** `&` *value*, *size_type idx*

  searches against the characters of *value*, starting with index *idx* in `*this`.
- `const` *char** *value*

  searches against the characters of the C-string *value*.
- `const` *char** *value*, *size_type idx*

  searches against the characters of the C-string *value*, starting with index *idx* in `*this`.
- `const` *char** *value*, *size_type idx*, *size_type value_len*

  searches against the *value_len* characters of the character array *value*, starting with index *idx* in `*this`. Thus, the null character (`'\0'`) has *no* special meaning here inside *value*.
- `const` *char value*

  searches against the character *value*.
- `const` *char value*, *size_type idx*

searches against the characters *value*, starting with index *idx* in `*this`.

For example:

Click here to view code image

```
std::string s("Hi Bill, I'm ill, so please pay the bill");

s.find("il")              // returns 4 (first substring "il")
s.find("il",10)           // returns 13 (first substring "il" starting from s[10])
s.rfind("il")             // returns 37 (last substring "il")
s.find_first_of("il")     // returns 1 (first char 'i' or 'l')
s.find_last_of("il")      // returns 39 (last char 'i' or 'l')
s.find_first_not_of("il") // returns 0 (first char neither 'i' nor 'l')
s.find_last_not_of("il")  // returns 36 (last char neither 'i' nor 'l')
s.find("hi")              // returns npos
```

Note that the naming scheme of the STL search algorithms differs from that for string search functions (see Section 11.2.2, page 507, for details).

13.2.12. The Value npos

If a search function fails, it returns *string* `::npos`. Consider the following example:

Click here to view code image

```
std::string s;
std::string::size_type idx;        // be careful: don't use any other type!
...
idx = s.find("substring");
if (idx == std::string::npos) {
    ...
}
```

The condition of the `if` statement yields `true` if and only if `"substring"` is not part of string `s`.

Be very careful when using the string value `npos` and its type. When you want to check the return value, always use `string::size_type`, *not* `int` or `unsigned` for the type of the return value; otherwise, the comparison of the return value with `string::npos` might not work. This behavior is the result of the design decision that `npos` is defined as `-1`:

Click here to view code image

```
namespace std {
     template <typename charT,
               typename traits = char_traits<charT>,
               typename Allocator = allocator<charT> >
     class basic_string {
       public:
         typedef typename Allocator::size_type size_type;
         ...
         static const size_type npos = -1;
         ...
     };
}
```

Unfortunately, `size_type`, which is defined by the allocator of the string, must be an unsigned integral type. The default allocator, `allocator`, uses type `size_t` as `size_type`. Because `-1` is converted into an unsigned integral type, `npos` is the maximum unsigned value of its type. However, the exact value depends on the exact definition of type `size_type`. Unfortunately, these maximum values differ. In fact, `(unsigned long)-1` *differs* from `(unsigned short)-1` if the size of the types differs. Thus, the comparison

```
idx == std::string::npos
```

might yield `false` if `idx` has the value `-1` and `idx` and `string::npos` have different types:

Click here to view code image

```
std::string s;
...
int idx = s.find("not found");     // assume it returns npos
if (idx == std::string::npos) {    // ERROR: comparison might not work
    ...
}
```

One way to avoid this error is to check whether the search fails directly:

```
if (s.find("hi") == std::string::npos) {
    ...
}
```

However, often you need the index of the matching character position. Thus, another simple solution is to define your own signed value for `npos` :

```
const int NPOS = -1;
```

Now the comparison looks a bit different and even more convenient:

```
if (idx == NPOS) {        // works almost always
    ...
}
```

Unfortunately, this solution is not perfect, because the comparison fails if either `idx` has type `unsigned short` or the index is greater than the maximum value of `int` . Because of these problems, the standard did not define it that way. However, because both might happen very rarely, the solution works in most situations. To write portable code, however, you should always use *string* `::size_type` for any index of your string type. For a perfect solution, you'd need some overloaded functions that consider the exact type of `string::size_type` . I still hope the standard will provide a better solution in the future (although with C++11 nothing changed).

13.2.13. Numeric Conversions

Since C++11, the C++ standard library provides convenience functions to convert strings into numeric values or to convert numeric values to strings (see Table 13.6). Note, however, that these conversions are available only for types `string` and `wstring` , not `u16string` and `u32string` .

Table 13.6. Numeric Conversions for Strings

String Function	Effect
stoi(*str*, *idxRet*=nullptr, *base*=10)	Converts *str* to an int
stol(*str*, *idxRet*=nullptr, *base*=10)	Converts *str* to a long
stoul(*str*, *idxRet*=nullptr, *base*=10)	Converts *str* to an unsigned long
stoll(*str*, *idxRet*=nullptr, *base*=10)	Converts *str* to a long long
stoull(*str*, *idxRet*=nullptr, *base*=10)	Converts *str* to an unsigned long long
stof(*str*, *idxRet*=nullptr)	Converts *str* to a float
stod(*str*, *idxRet*=nullptr)	Converts *str* to a double
stold(*str*, *idxRet*=nullptr)	Converts *str* to a long double
to_string(*val*)	Converts *val* to a string
to_wstring(*val*)	Converts *val* to a wstring

For all function that convert strings to a numeric value, the following applies:

- They skip leading whitespaces.
- They allow you to return the index of the first character after the last processed character.
- They might throw `std::invalid_argument` if no conversion is possible and `std::out_of_range` if the converted value is outside the range of representable values for the return type.
- For integral values, you can optionally pass the number base to use.

For all functions that convert a numeric value to a `string` or `wstring` , *val* may be any of the following types: `int` , `unsigned int` , `long` , `unsigned long` , `long long` , `unsigned long long` , `float` , `double` , or `long double` .

For example, consider the following program:

Click here to view code image

```
// string/stringnumconv1.cpp

#include <string>
#include <iostream>
#include <limits>
#include <exception>
```

```
int main()
{
    try {
      // convert to numeric type
      std::cout << std::stoi ("  77") << std::endl;
      std::cout << std::stod ("  77.7") << std::endl;
      std::cout << std::stoi ("-0x77") << std::endl;

     // use index of characters not processed
     std::size_t idx;
     std::cout << std::stoi ("  42 is the truth", &idx) << std::endl;
     std::cout << " idx of first unprocessed char: " << idx <<
std::endl;

     // use bases 16 and 8
     std::cout << std::stoi ("  42", nullptr, 16) << std::endl;
     std::cout << std::stol ("789", &idx, 8) << std::endl;
     std::cout << " idx of first unprocessed char: " << idx <<
std::endl;

     // convert numeric value to string
     long long ll = std::numeric_limits<long long>::max();
     std::string s = std::to_string(ll);  // converts maximum long long to
string
     std::cout << s << std::endl;

     // try to convert back
     std::cout << std::stoi(s) << std::endl;  // throws out_of_range
    }
    catch (const std::exception& e) {
      std::cout << e.what() << std::endl;
    }
}
```

The program has the following output:

```
77
77.7
0
42
 idx of first unprocessed char: 4
66
7
 idx of first unprocessed char: 1
9223372036854775807
stoi argument out of range
```

Note that `std::stoi("-0x77")` yields `0` because it parses only `-0`, interpreting the `x` as the end of the numeric value found. Note that `std::stol("789",&idx,8)` parses only the first character of the string because `8` is not a valid character for octal numbers.

13.2.14. Iterator Support for Strings

A string is an ordered collection of characters. As a consequence, the C++ standard library provides an interface for strings that lets you use them as STL containers.[10]

[10] The STL is introduced in Chapter 6.

In particular, you can call the usual member functions to get iterators that iterate over the characters of a string. If you are not familiar with iterators, consider them as something that can refer to a single character inside a string, just as ordinary pointers do for C-strings. By using these objects, you can iterate over all characters of a string by calling several algorithms that either are provided by the C++ standard library or are user defined. For example, you can sort the characters of a string, reverse the order, or find the character that has the maximum value.

String iterators are random-access iterators. This means that they provide random access, and you can use all algorithms (see Section 6.3.2, page 198, and Section 9.2, page 433, for a discussion about iterator categories). As usual, the types of string iterators (`iterator` , `const_iterator` , and so on) are defined by the string class itself. The exact type is implementation defined, but string iterators are often defined simply as ordinary pointers. See Section 9.2.6, page 440, for a nasty difference between iterators that are implemented as pointers and iterators that are implemented as classes.

Iterators are invalidated when reallocation occurs or when certain changes are made to the values to which they refer. See Section 13.2.6, page 672, for details.

Iterator Functions for Strings

Table 13.7 shows all the member functions that strings provide for iterators. As usual, the range specified by `beg` and `end` is a half-open range that includes `beg` but excludes `end`, written as [`beg` , `end`) (see Section 6.3, page 188).

Table 13.7. Iterator Operations of Strings

Expression	Effect
`s.begin(), s.cbegin()`	Returns a random-access iterator for the first character
`s.end(), s.cend()`	Returns a random-access iterator for the position after the last character
`s.rbegin(), s.crbegin()`	Returns a reverse iterator for the first character of a reverse iteration (thus, for the last character)
`s.rend(), crend()`	Returns a reverse iterator for the position after the last character of a reverse iteration (thus, the position before the first character)
`string s(`*beg*`,`*end*`)`	Creates a string that is initialized by all characters of the range [*beg*,*end*)
`s.append(`*beg*`,`*end*`)`	Appends all characters of the range [*beg*,*end*)
`s.assign(`*beg*`,`*end*`)`	Assigns all characters of the range [*beg*,*end*)
`s.insert(`*pos*`,`*c*`)`	Inserts the character *c* at iterator position *pos* and returns the iterator position of the new character
`s.insert(`*pos*`,`*num*`,`*c*`)`	Inserts *num* occurrences of the character *c* at iterator position *pos* and returns the iterator position of the first new character
`s.insert(`*pos*`,`*beg*`,`*end*`)`	Inserts all characters of the range [*beg*,*end*) at iterator position *pos*
`s.insert(`*pos*`,`*initlist*`)`	Inserts all characters of the initializer list *initlist* at iterator position *pos* (since C++11)
`s.erase(`*pos*`)`	Deletes the character to which iterator *pos* refers and returns the position of the next character
`s.erase(`*beg*`,`*end*`)`	Deletes all characters of the range [*beg*,*end*) and returns the next position of the next character
`s.replace(`*beg*`,`*end*`,`*str*`)`	Replaces all characters of the range [*beg*,*end*) with the characters of string *str*
`s.replace(`*beg*`,`*end*`,`*cstr*`)`	Replaces all characters of the range [*beg*,*end*) with the characters of the C-string *cstr*
`s.replace(`*beg*`,`*end*`,`*cstr*`,`*len*`)`	Replaces all characters of the range [*beg*,*end*) with *len* characters of the character array *cstr*
`s.replace(`*beg*`,`*end*`,`*num*`,`*c*`)`	Replaces all characters of the range [*beg*,*end*) with *num* occurrences of the character *c*
`s.replace(`*beg*`,`*end*`,` *newBeg*`,`*newEnd*`)`	Replaces all characters of the range [*beg*,*end*) with all characters of the range [*newBeg*,*newEnd*)
`s.replace(`*beg*`,`*end*`,` *initlist*`)`	Replaces all characters of the range [*beg*,*end*) with the values of the initializer list *initlist* (since C++11)

To support the use of back inserters for strings, the `push_back()` function is defined. See Section 9.4.2, page 455, for details about back inserters and page 688 for an example of their use with strings.

Example of Using String Iterators

A very useful thing that you can do with string iterators is to make all characters of a string lowercase or uppercase via a single statement. For example:

Click here to view code image

```
// string/stringiter1.cpp

#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
#include <regex>
using namespace std;

int main()
```

```
{
    // create a string
    string s("The zip code of Braunschweig in Germany is 38100");
    cout << "original: " << s << endl;

    // lowercase all characters
    transform (s.cbegin(), s.cend(),    // source
               s.begin(),               // destination
               [] (char c) {            // operation
                   return tolower(c);
               });
    cout << "lowered:  " << s << endl;

    // uppercase all characters
    transform (s.cbegin(), s.cend(),    // source
               s.begin(),               // destination
               [] (char c) {            // operation
                   return toupper(c);
               });
    cout << "uppered:  " << s << endl;

    // search case-insensitive for Germany
    string g("Germany");
    string::const_iterator pos;
    pos = search (s.cbegin(),s.cend(),      // source string in which to search
                  g.cbegin(),g.cend(),      // substring to search
                  [] (char c1, char c2) {   // comparison criterion
                      return toupper(c1) == toupper(c2);
                  });
    if (pos != s.cend()) {
        cout << "substring \"" << g << "\" found at index "
             << pos - s.cbegin() << endl;
    }
}
```

Here, we twice use iterators provided by `cbegin()`, `cend()`, and `begin()` to pass them to the `transform()` algorithm, which transforms all elements of an input range to a destination range by using a transformation passed as fourth argument (see Section 6.8.1, page 225, and Section 11.6.3, page 563, for details).

The transformation is specified with a lambda (see Section 6.9, page 229), which converts the elements of the string (the characters) to lower- or uppercase. Note that `tolower()` and `toupper()` are old C functions that use the global locale. If you have a different locale or more than one locale in your program, you should use the new form of `tolower()` and `toupper()`. See Section 16.4.4, page 895, for details.

Finally, we use the search algorithm to search for a substring with our own search criterion. This criterion is a lambda that compares the characters in a case-insensitive way.

Alternatively, we could use the regex library:

Click here to view code image

```
// search case-insensitive for Germany
std::regex pat("Germany",                   // expression to search for
               regex_constants::icase);     // search case-insensitive
smatch m;
if (regex_search (s,m,pat)) {               // search regex pattern in s
    cout << "substring \"Germany\" found at index "
         << m.position() << endl;
}
```

See Section 14.6, page 732, for details.

Thus, the output of the program is as follows:

```
original: The zip code of Braunschweig in Germany is 38100
lowered:  the zip code of braunschweig in germany is 38100
uppered:  THE ZIP CODE OF BRAUNSCHWEIG IN GERMANY IS 38100
substring "Germany" found at index 32
```

In the last output statement, you can process the difference of two string iterators to get the index of the character position:

```
pos - s.cbegin()
```

You can use operator `-` because string iterators are random-access iterators. Similar to transferring an index into the iterator position, you can simply add the value of the index.

If you use strings in sets or maps, you might need a special sorting criterion to let the collections sort the string in a case-insensitive way. See Section 7.8.6, page 351, for an example that demonstrates how to do this.

The following program demonstrates other examples of strings using iterator functions:

Click here to view code image

```
// string/stringiter2.cpp

#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    // create constant string
    const string hello("Hello, how are you?");

    // initialize string s with all characters of string hello
    string s(hello.cbegin(),hello.cend());

    // ranged-based for loop that iterates through all the characters
    for (char c : s) {
        cout << c;
    }
    cout << endl;

    // reverse the order of all characters inside the string
    reverse (s.begin(), s.end());
    cout << "reverse:       " << s << endl;

    // sort all characters inside the string
    sort (s.begin(), s.end());
    cout << "ordered:       " << s << endl;

    // remove adjacent duplicates
    // - unique() reorders and returns new end
    // - erase() shrinks accordingly
    s.erase (unique(s.begin(),
                    s.end()),
             s.end());
    cout << "no duplicates: " << s << endl;
}
```

The program has the following output:

```
Hello, how are you?
reverse:       ?uoy era woh ,olleH
ordered:          ,?Haeehlloooruwy
no duplicates:  ,?Haehloruwy
```

The following example uses back inserters to read the standard input into a string:

Click here to view code image

```
// string/string3.cpp

#include <string>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <locale>
using namespace std;

int main()
{
    string input;

    // don't skip leading whitespaces
    cin.unsetf (ios::skipws);

    // read all characters while compressing whitespaces
    const locale& loc(cin.getloc());     // locale
    unique_copy(istream_iterator<char>(cin),  // beginning of source
                istream_iterator<char>(),     // end of source
                back_inserter(input),         // destination
                [=] (char c1, char c2) {      // criterion for adj. duplicates
                    return isspace(c1,loc) && isspace(c2,loc);
                });

    // process input
    // - here: write it to the standard output
    cout << input;
```

```
}
```

By using the `unique_copy()` algorithm (see Section 11.7.2, page 580), all characters are read from the input stream `cin` and inserted into the string `input`.

The passed lambda operation checks whether two characters are whitespaces. This criterion is taken by `unique_copy()` to detect adjacent "duplicates," where the second element can be removed. Thus, while reading the input, the algorithm compresses multiple whitespaces (see Section 16.4.4, page 895, for a discussion of `isspace()`).

The criterion itself takes the current local into account. To do this, `loc` is initialized by the locale of `cin` and passed by value to the lambda (see Section 15.8, page 790, for details of `getloc()`).

You can find a similar example in the reference section about `unique_copy()` in Section 11.7.2, page 582.

13.2.15. Internationalization

As mentioned in Section 13.2.1, page 664, the template string class `basic_string<>` is parametrized by the character type, the traits of the character type, and the memory model. Type `string` is the specializations for characters of type `char`, whereas types `wstring`, `u16string`, and `u32string` are the specializations for characters of type `wchar_t`, `char16_t`, and `char32_t`, respectively.

Note that you can specify the character sets used for string literals since C++11 (see Section 3.1.6, page 23).

To specify the details of how to deal with aspects depending on the representation of a character type, character traits are provided. An additional class is necessary because you can't change the interface of built-in types, such as `char` and `wchar_t`, and the same character type may have different traits. The details about the traits classes are described in Section 16.1.4, page 853.

The following code defines a special traits class for strings so that they operate in a case-insensitive way:

Click here to view code image

```
// string/icstring.hpp

#ifndef ICSTRING_HPP
#define ICSTRING_HPP

#include <string>
#include <iostream>
#include <cctype>

// replace functions of the standard char_traits<char>
// so that strings behave in a case-insensitive way
struct ignorecase_traits : public std::char_traits<char> {
    // return whether c1 and c2 are equal
    static bool eq(const char& c1, const char& c2) {
        return std::toupper(c1)==std::toupper(c2);
    }
    // return whether c1 is less than c2
    static bool lt(const char& c1, const char& c2) {
        return std::toupper(c1)<std::toupper(c2);
    }
    // compare up to n characters of s1 and s2
    static int compare(const char* s1, const char* s2,
                       std::size_t n) {
        for (std::size_t i=0; i<n; ++i) {
            if (!eq(s1[i],s2[i])) {
                return lt(s1[i],s2[i])?-1:1;
            }
        }
        return 0;
    }
    // search c in s
    static const char* find(const char* s, std::size_t n,
                            const char& c) {
        for (std::size_t i=0; i<n; ++i) {
            if (eq(s[i],c)) {
                return &(s[i]);
            }
        }
        return 0;
    }
};

// define a special type for such strings
typedef std::basic_string<char,ignorecase_traits> icstring;
```

```
// define an output operator
// because the traits type is different from that for std::ostream
inline
std::ostream& operator << (std::ostream& strm, const icstring& s)
{
    // simply convert the icstring into a normal string
    return strm << std::string(s.data(),s.length());
}

#endif    // ICSTRING_HPP
```

The definition of the output operator is necessary because the standard defines I/O operators only for streams that use the same character and traits type. But here the traits type differs, so we have to define our own output operator. For input operators, the same problem occurs.

The following program demonstrates how to use these special kinds of strings:

Click here to view code image

```
// string/icstring1.cpp

#include "icstring.hpp"

int main()
{
    using std::cout;
    using std::endl;

    icstring s1("hallo");
    icstring s2("otto");
    icstring s3("hALLo");

    cout << std::boolalpha;
    cout << s1 << " == " << s2 << " : " << (s1==s2) << endl;
    cout << s1 << " == " << s3 << " : " << (s1==s3) << endl;

    icstring::size_type idx = s1.find("All");
    if (idx != icstring::npos) {
        cout << "index of \"All\" in \"" << s1 << "\": "
             << idx << endl;
    }
    else {
        cout << "\"All\" not found in \"" << s1 << endl;
    }
}
```

The program has the following output:

```
hallo == otto : false
hallo == hALLo : true
index of "All" in "hallo": 1
```

See Chapter 16 for more details about internationalization.

13.2.16. Performance

As usual, the standard does *not* specify *how* the string class is to be implemented but instead specifies only the interface. There may be important differences in speed and memory usage, depending on the concept and priorities of the implementation.

Note that since C++11, reference counted implementations are not permitted any longer. The reason is that an implementation that lets strings share internal buffers doesn't work in multithreaded contexts.

13.2.17. Strings and Vectors

Strings and vectors behave similarly. This is no surprise because both are containers that are typically implemented as dynamic arrays. Thus, you could consider a string as a special kind of a vector that has characters as elements. In fact, you can use a string as an STL container (see Section 13.2.14, page 684). However, considering a string as a special kind of vector is dangerous because there are many fundamental differences between the two. Chief among these are their two primary goals:

1. The primary goal of vectors is to handle and to manipulate the elements of the container, not the container as a whole. Thus, vector implementations are optimized to operate on elements inside the container.
2. The primary goal of strings is to handle and to manipulate the container (the string) as a whole. Thus, strings are optimized to reduce the costs of assigning and passing the whole container.

These different goals typically result in completely different implementations. Nevertheless, you can also use vectors as ordinary C-strings. See Section 7.3.3, page 278, for details.