

std::string Quick Reference Card

Last Revised: August 18, 2013

© Copyright 2013 by Peter Chapin

Permission is granted to copy and distribute freely, for any purpose, provided the copyright notice above is not modified. Permission is also granted to edit this document to change its file format.

- *The type `string::size_type` is an unsigned integral type for use as an index or as a length.* The special value `string::npos`, of type `string::size_type`, can never be used as a valid index.
- *The size of a string is the number of characters in the string.* The capacity of a string is the number of character sized units of memory reserved by the string. The capacity is always greater than or equal to the size.
- *Substrings are defined by an index to the first character and a length.* If the length requested for a substring is larger than the number of characters remaining in the string, all of the remaining characters are taken. A substring length of `string::npos` requests the rest of the string no matter how many characters are remaining. If the starting index is greater than the string's size, a `std::out_of_range` exception is thrown. If the starting index is equal to the string's size the only substring is the empty string.
- *Strings can contain binary data.* The null character is not treated in any special way in a string.

Constructors (and related methods)

<code>string();</code>	Constructs an empty string.
<code>string(const string &str, size_type pos = 0, size_type n = npos);</code>	Copies <code>str</code> or a substring of <code>str</code> .
<code>string(const char *s);</code>	Copies a c-string.
<code>string(size_type n, char c);</code>	Constructs a string by making <code>n</code> copies of <code>c</code> .
<code>string &operator=(const string &str);</code>	Assigns <code>str</code> to the current object.
<code>string &operator=(const char *s);</code>	Assigns a c-string to the current object.
<code>string &assign(const string &str, size_type pos, size_type n);</code>	Assigns a substring of <code>str</code> to the current object.
<code>string &assign(size_type n, char c);</code>	Assigns a string of <code>n</code> copies of <code>c</code> to the current object.

Adding Characters

<pre>string &operator+=(const string &str);</pre>	Appends <code>str</code> to the current object.
<pre>string &operator+=(const char *s);</pre>	Appends a c-string to the current object.
<pre>string &operator+=(char c);</pre>	Appends the character <code>c</code> to the current object.
<pre>string &append(const string &str, size_type pos, size_type n);</pre>	Appends a substring of <code>str</code> to the current object.
<pre>string &append(size_type n, char c);</pre>	Appends <code>n</code> copies of <code>c</code> to the current object.
<pre>string &insert(size_type pos1, const string &str);</pre>	Inserts <code>str</code> into current object at position <code>pos1</code> .
<pre>string &insert(size_type pos1, const char *s);</pre>	Inserts a c-string into current object at position <code>pos1</code> .
<pre>string &insert(size_type pos1, size_type n, char c);</pre>	Inserts <code>n</code> copies of <code>c</code> into current object at <code>pos1</code> .
<pre>string &insert(size_type pos1, const string &str, size_type pos2, size_type n);</pre>	Inserts substring of <code>str</code> into current object at <code>pos1</code> .

Removing Characters

<pre>string &erase(size_type pos = 0, size_type n = npos);</pre>	Erases a substring of the current object.
--	---

There are also a number of **replace** methods that take a `pos1` and `n1` as their first two parameters that define a substring of the current object. They then follow the same pattern as the insert member functions to specify the source text for the replacement.

Accessing Characters

<pre>char &operator[](size_type pos);</pre>	You can index a string with the <code>[]</code> operator. No bounds checking is done (faster).
<pre>char &at(size_type pos);</pre>	Similar to <code>operator[]</code> except that a <code>std::out_of_range</code> exception is thrown if <code>pos</code> is out of range (slower).

<pre>string substr(size_type pos = 0, size_type n = npos);</pre>	Returns a substring of the current object.
<pre>const char *c_str();</pre>	Returns a pointer to a c-style string containing the current object's contents.

Searching for Characters

<pre>size_type find(const string &str, size_type pos = 0);</pre>	Searches for first occurrence of <code>str</code> in the current object starting at <code>pos</code> . Returns position or <code>npos</code> if not found.
<pre>size_type find(const char *s, size_type pos = 0);</pre>	Searches for first occurrence of c-string <code>s</code> in the current object starting at <code>pos</code> . Returns position or <code>npos</code> if not found.
<pre>size_type find(char c, size_type pos = 0);</pre>	Searches for first occurrence of <code>c</code> in the current object starting at <code>pos</code> . Returns position or <code>npos</code> if not found.
<pre>size_type find_first_of(const string &str, size_type pos = 0);</pre>	Searches for the first occurrence of <i>any</i> character in <code>str</code> in the current object starting at <code>pos</code> . Returns position or <code>npos</code> if none found.
<pre>size_type find_first_of(const char *s, size_type pos = 0);</pre>	Searches for the first occurrence of <i>any</i> character in <code>s</code> in the current object starting at <code>pos</code> . Returns position or <code>npos</code> if none found.

There are also several `rfind` methods that work like the `find` methods above except that they search for the last occurrence instead of the first. The default value for `pos` for those methods is `npos`.

There are also two `find_first_not_of` methods that work like the `find_first_of` methods except that they search for the first occurrence of any character that is *not* in the given string.

Finally there are two `find_last_of` and `find_last_not_of` methods that work like the `find_first_of` and `find_first_not_of` methods except that they search for the last occurrence of any character in (or not in) the given string. The default value of `pos` for those functions is `npos`.

Useful Free Functions

<pre>string operator+(const string &lhs, const string &rhs);</pre>	Concatenates the given strings and returns the result as a new string.
--	--

<pre>string operator+(const char *lhs, const string &rhs); string operator+(const string &lhs, const char *rhs);</pre>	Concatenates a c-string and a string and returns the result as a new string.
<pre>string operator+(char lhs, const string &rhs); string operator+(const string &lhs, char rhs);</pre>	Concatenates a character and a string and returns the result as a new string.
<pre>bool operator==(const string &lhs, const string &rhs);</pre>	Compares the two strings. Returns true if they are equal.
<pre>void swap(string &lhs, string &rhs);</pre>	Swaps two strings. This operation is optimized so that it only requires a (short) time that is unrelated to the size of the strings involved.

All the other relational operators (!=, <, >, <=, >=) are also supported. Furthermore overloaded relational operators exists that allow for comparisons directly with c-strings (on either the left or right hand sides). Comparing strings to characters directly is not supported.

Memory Management Functions

<pre>size_type size(); size_type length();</pre>	Returns the number of characters in the current object.
<pre>size_type capacity();</pre>	Returns the number of characters the current object can hold without reallocating storage.
<pre>void resize(size_type n);</pre>	Sets the size to n. If n is less than the current size, characters are lost. If n is greater than the current size, the new characters are initialized with the null character.
<pre>void resize(size_type n, char c);</pre>	Similar to <code>resize(size_type)</code> except that c is used to initialize new characters in the case where the size is expanded.
<pre>void reserve(size_type n);</pre>	Increase capacity to at least n. By making this call before extending the size of a string, you can greatly enhance the string's memory management efficiency.

String I/O Operations (non members)

<pre>ostream &operator<<(ostream &os, const string &str);</pre>	Outputs str to the given output stream.
<pre>istream &operator>>(istream &is, string &str);</pre>	Inputs a white space delimited word of any length from the given input stream into str.

<pre>istream &getline(istream &is, string &str);</pre>	<p>Inputs a line of any length from the given input stream into <code>str</code>. The line ends at the first <code>'\n'</code> encountered or when the stream reaches EOF. The <code>'\n'</code> is removed from the stream, but not added to the string.</p>
<pre>istream &getline(istream &is, string &str, char delim);</pre>	<p>Similar to the <code>getline</code> above except that <code>delim</code> is used to delimit the lines instead of <code>'\n'</code>.</p>

Container Functions

Strings allow themselves to be accessed and manipulated like standard containers. They provide a `string::iterator` type and methods **begin** and **end** for creating appropriate iterators. String iterators are in the random access category. Strings also provide a **push_back** method for appending characters to the end, and several iterator-based insert and searching functions. In this respect `std::string` is similar to `std::vector<char>`. These functions are not detailed in this version of this quick reference card.