std::string Quick Reference Card

Last Revised: August 18, 2013

- *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)

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

Adding Characters

`string &operator+=( const string &str );`	Appends `str` to the current object.
`string &operator+=( const char *s );`	Appends a c-string to the current object.
`string &operator+=( char c );`	Appends the character `c` to the current object.

`string &append(`
`  const string &str,`
`  size_type pos,`
`  size_type n );`
	Appends a substring of `str` to the current object.

`string &append( size_type n, char c );`	Appends `n` copies of `c` to the current object.

`string &insert(`
`  size_type pos1, const string &str );`
	Inserts `str` into current object at position `pos1`.

`string &insert(`
`  size_type pos1, const char *s );`
	Inserts a c-string into current object at position `pos1`.

`string &insert(`
`  size_type pos1, size_type n, char c );`
	Inserts `n` copies of `c` into current object at `pos1`.

`string &insert(`
`  size_type pos1,`
`  const string &str,`
`  size_type pos2,`
`  size_type n );`
	Inserts substring of `str` into current object at `pos1`.

Removing Characters

`string &erase(`
`  size_type pos = 0, size_type n = npos );`
	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

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

`string substr(`
`  size_type pos = 0,`
`  size_type n = npos );`
	Returns a substring of the current object.

`const char *c_str( );`	Returns a pointer to a c-style string containing the current object's contents.

Searching for Characters

`size_type find(`
`  const string &str,`
`  size_type pos = 0 );`
	Searches for first occurrence of `str` in the current object starting at `pos`. Returns position or `npos` if not found.

`size_type find(`
`  const char *s, size_type pos = 0 );`
	Searches for first occurrence of c-string `s` in the current object starting at `pos`. Returns position or `npos` if not found.

`size_type find(`
`  char c, size_type pos = 0 );`
	Searches for first occurrence of `c` in the current object starting at `pos`. Returns position or `npos` if not found.

`size_type find_first_of(`
`  const string &str,`
`  size_type pos = 0 );`
	Searches for the first occurrence of *any* character in `str` in the current object starting at `pos`. Returns position or `npos` if none found.

`size_type find_first_of(`
`  const char *s,`
`  size_type pos = 0 );`
	Searches for the first occurrence of *any* character in `s` in the current object starting at `pos`. Returns position or `npos` 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

`string operator+(const string &lhs,`
`const string &rhs);`
	Concatenates the given strings and returns the result as a new string.

`string operator+(`
`  const char *lhs, const string &rhs );`
`string operator+(`
`  const string &lhs, const char *rhs );`
	Concatenates a c-string and a string and returns the result as a new string.

`string operator+(`
`  char lhs, const string &rhs );`
`string operator+(`
`  const string &lhs, char rhs );`
	Concatenates a character and a string and returns the result as a new string.

`bool operator==(`
`  const string &lhs,`
`  const string &rhs );`
	Compares the two strings. Returns true if they are equal.

`void swap( string &lhs, string &rhs );`	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

`size_type size( );`
`size_type length( );`
	Returns the number of characters in the current object.

`size_type capacity( );`	Returns the number of characters the current object can hold without reallocating storage.

`void resize(`
`  size_type n );`
	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.

`void resize(`
`  size_type n,`
`  char c );`
	Similar to `resize(size_type)` except that `c` is used to initialize new characters in the case where the size is expanded.

`void reserve(`
`  size_type n );`
	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)

`ostream &operator<<(`
`  ostream &os,`
`  const string &str );`
	Outputs `str` to the given output stream.

`istream &operator>>(`
`  istream &is,`
`  string &str );`
	Inputs a white space delimited word of any length from the given input stream into `str`.

`istream &`**`getline`**`(`
`  istream &is,`
`  string &str );`
	Inputs a line of any length from the given input stream into `str`. The line ends at the first `'\n'` encountered or when the stream reaches EOF. The `'\n'` is removed from the stream, but not added to the string.

`istream &`**`getline`**`(`
`  istream &is,`
`  string &str,`
`  char delim );`
	Similar to the `getline` above except that `delim` is used to delimit the lines instead of `'\n'`.

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.