# C++ and CS106 Library Reference

Written by Julie Zelenski and revised by Jerry.

A couple of decent C++ web resources you might want to bookmark:

```
http://www.cppreference.com
http://www.cplusplus.com/ref/
http://msdn2.microsoft.com/en-us/library/cscc687y.aspx
```

These can be useful for anything in standard C++, which includes the language itself and all of its standard libraries (**`string`**, **`stream`**, **`ctype`**, **`math`**, etc.) The Stanford-specific libraries are also documented very nicely, and that documentation can be viewed by following the **CS106X Library Documentation** link in the **CS106X Resources** section of the course web site.

### The standard C++ `string` class

The string class is defined in **`<string>`**. The **`string`** type is actually a **`typedef`** shorthand. The underlying full name is

```
std::basic_string<char, std::char_traits<char>, std::allocator<char>>.
```

You don't need to worry about this sort of low-level goop, but you will see the full name in compiler error messages and will want to recognize it as such.

The default constructor initializes a string variable to the empty string, thus declaring a string variable ensures that its contents start empty. This is unlike the built-in types (**`int`**, **`double`**, etc.) that have random contents until explicitly initialized. Assigning one string to another via **`=`** or passing/returning a string makes a new distinct copy of the same character sequence. Strings are mutable, unlike Java strings.

A string literal, i.e., sequence of characters within double-quotes such as **`"binky"`**, is actually an old-style C-string. You can typically use a C-string wherever a string object is required since there is an automatic conversion from C-string to new-style C++ string object. If ever need to force this conversion, you can do so using a syntax similar to a typecast: **`string("binky").`** This is invoking the string class constructor that takes a C-string argument.

In general, operations on strings are designed to be very efficient and, as a result, some do not check parameters for validity. It is the client's job to ensure positions/lengths are in bounds for calls to **`substr`**, find, replace, and so on. The behavior on incorrect calls is implementation-dependent, but unlikely to be pleasant in any situation.

| | |
|---|---|
| *str*.**length()**<br>*str*.**size()** | Returns number of characters in receiver string (**length** and **size** are synonyms) |
| *str*[*index*]<br>*str*.**at(***index***)** | Access character at specified **index** in receiver string. Indexes start at 0. **at** throws an exception if out of bounds, operator[] does not bounds-check (for efficiency). |
| *str*.**empty()** | Returns true if receiver string is equal to "", false otherwise |
| *str1* **+** *str2*<br>*str1* **+** *ch* | **+** is overloaded to allow strings to be concatenated with other strings and single chars. The result is a new string containing concatenation of the operands. |
| *str*.**find(***key, pos***)** | Searches for **key** (which can be either string or single character) within receiver string, starting search at index **pos**. If **pos** not specified, default value of 0 is used. Returns index of key if found or **string::npos** otherwise. |
| *str*.**substr(***pos, len***)** | Returns a new string containing **len** chars starting from index **pos** in receiver string. If **len** is not given, takes all characters to end of string. |
| *str*.**insert** (*pos, text*) | Inserts **text** starting at index **pos** into the receiver string. Modifies receiver string. |
| *str*.**replace** (*pos, count, text*) | Removes **count** chars from receiver string starting at index **pos**, and replaces with **text**. Modifies receiver string. |
| *str1* **<** *str2*<br>**== != < > <= >=** | String comparison uses standard relational operators. Ordering is lexicographic (dictionary ordering) and case-sensitive. |
| *str*.**c_str** () | Returns receiver string in old-style C-string form. Used when you need backward compatibility with an older function. |

### CS106 string utility functions

**strlib.h** contains a few conveniences for handling string conversions. These are free functions (i.e. not member functions invoked on a receiver string).

| | |
|---|---|
| **realToString**(*d*)<br>**stringToReal**(*str*) | Convert double value to string form and vice versa. **stringToReal** raises an error if string is not well-formed. |
| **integerToString**(*i*)<br>**stringToInteger**(*str*) | Convert integer value to string form and vice versa. **stringToInteger** raises an error if string is not well-formed. |
| **toUpperCase**(*s*)<br>**toLowerCase**(*s*) | Returns a new string, which is a copy of input string where all alphabetic characters have been converted to upper/lower case equivalents, non-letter characters are unchanged. |
| **equalsIgnoreCase**(*s*, *t*) | Returns **true** if and only if **s** and **t** are the same string, minus lowercase/uppercase distinctions. Whereas **"ab" != "AB"**, **equalsIgnoreCase("ab", "AB")** would return **true**. |
| **startsWith**(*s*, *t*)<br>**endsWith**(*s*, *t*) | Returns **true** if and only if the string **s** begins with (or ends with) the string or the character **t**. So, **startsWith("abc", "ab")** would return **true**, whereas **startsWith("abcdef", "abcf")** would return **false**. |
| **trim**(*s*) | Returns a copy of the string **s**, except that all leading and trailing whitespace has been removed. |

### Standard C++ stream classes

The global streams **cin**/**cout** and the basic stream classes are defined in **<iostream>.** The file stream classes are defined in **<fstream>.** There are many variants of stream classes in the standard library, we typically will use **ifstream** for input file streams, and **ofstream** for output file streams. There are many more features available on streams than I will list here. I/O isn't particularly interesting to study and we will mostly just use the simple features, so no need to dig deep.

Like strings, the stream class names are also shortened with a **typedef**. The full, underlying name for **ifstream** is **std::basic_ifstream<char, std::char_traits<char>>** and **ofstream** is same with **ofstream** substituted for **ifstream**.

Copying of stream objects is discouraged. Streams should typically be passed by reference. In most library implementations, copying a stream (either from direct assignment or pass-by-value) is specifically disallowed and will not compile.

These member functions apply to both input and output streams:

| | |
|---|---|
| *stream*.**open(***filenameAsCString***)** | Opens named file and attaches to receiver stream. If unsuccessful, sets stream error state. The filename parameter is expected to be an old-style C-string! (see **c_str** above for how to convert a C++ string to C-string) |
| *stream*.**close()** | Closes file. This is automatically done by stream destructor, but if you open another file on the stream, you first explicitly close any open one. |
| *stream*.**fail()** | Returns true if the receiver stream is in an error state, e.g a previous stream operation was not successful. Once a stream gets into an error state, the error state persists and no further operations on that stream can succeed until the error state is cleared (see **clear** below) |
| *stream*.**clear()** | Clears error state of the receiver stream |

These operations are specific to output streams.

| | |
|---|---|
| *ostream* **<<** *num* **<<** *str* **<<** *ch* | Stream insertion **<<** does formatted output. See **<iomanip>** for all the fancy features for controlling width/precision/alignment/format. |
| *ostream*.**put(***ch***)** | Outputs a single char onto receiver stream |

These operations are specific to input streams.

| | |
|---|---|
| *istream* **>>** *num* **>>** *str* **>>** *ch* | Stream extraction **>>** reads formatted input. By default, skips white space. Puts stream into fail state if read doesn't match expected. |
| *istream*.**peek()**<br>*istream*.**get()** | Read next character from receiver stream. Return **EOF** (–1) if no more characters to read. Returns an **int** rather than **char** because of need to represent EOF. **peek** returns the next character but doesn't remove it from the stream |

| | |
|---|---|
| *istream*.**unget()** | Pushes last character read back onto the receiver stream |
| **getline**(istream & in, string & str, char delimiter = '\n') | Reads next line of input (up to **delimiter**) and stores in **str** reference parameter. **Note:** this is a free function not a stream member function! You pass the stream to read from as the first argument. |

**CS106 simple input functions**

Handling user input can be a little messy (i.e. retrying on errors, etc.), so these simplified input routines are provided in our **simpio.h** to make your life a little easier. These are supplied as free functions.

| | |
|---|---|
| string **getLine**(*prompt*)<br>int **getInteger**(*prompt*)<br>long **getLong**(*prompt*)<br>double **getReal**(*prompt*) | Each prompts the user with the specified **prompt**, reads a line of input from the user and returns the value. In case of the numeric versions, if user's input is not well-formed, re-prompts and tries again until input is valid. The **prompt** may be omitted if no prompt is needed. |

**CS106 random library**

**random.h** contains a set of functions that generate pseudo-random events. The implementation is layered on top of the standard C functions **rand**/**srand** from **<cstdlib>**.

| | |
|---|---|
| void **setRandomSeed**(*seed*) | Seeds random number generator. |
| int **randomInteger**(*low*, *high*)<br>double **randomReal**(*low*, *high*) | Returns **int**/real from random range. |
| bool **randomChance**(*probability*) | Returns **true**/**false** based on random probability. |

**Advanced Libraries**

There are more advanced libraries that aren't being outlined here, because we'll be learning them piecemeal over the course of the next several weeks.