Library quick reference

You're sitting at your computer, trying to write some code that manipulates a string or stream or draws some graphics. You know the functionality you need probably exists in one of the libraries, but don't remember the exact details. The information might be in the reader, but it's easy to get lost in there. You could open the header file and check it out directly. This works okay for our CS106 interfaces (which tend to be nicely commented :-), but it is not so helpful for the standard headers which are often cryptic. The web, of course, is filled with gobs of information, both handy and useless, so you can probably find what you need there but you'll need to dig around some first.

So it seemed worthwhile to pull together a quick summary of the most commonly used functionality for handy reference. And thus, this document was born… This is a work-in-progress, so let me know if you note errors/omissions I should address in the future. Note that this is intended as a quick high-level overview, so if you need more in-depth details, check the reader/header file/web.

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 sites contain information about standard C++, which includes the language itself and all of its standard libraries (string, stream, ctype, math, etc.)

The CS106 library header files have been turned into web pages via Doxygen and are browsable online from our class web site via the "Documentation" link.

The standard C++ `string` class

The **`string`** class is defined in the header file **`<string>`**. The name **`string`** is actually a typedef shorthand. The underlying full name is **`std::basic_string<char, std::char_traits<char>, std::allocator<char>>`**. You needn't worry about the low-level goop, but you will see the full name in compiler error messages and will want to recognize it as such.

`string` is a class, and thus string variables are objects, each manages its own sequence of characters. The default constructor initializes a string variable to the empty string, thus simply declaring a string variable ensures that its contents start empty. This is unlike the primitive 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 string. You can retrieve or assign individual characters using square brackets. There are member functions to search a string for characters/substrings, extract a substring, modify the string by inserting/removing/replacing characters, and more. (see table on next page)

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 invokes the string class constructor on the C-string and initializes a string object from those characters.

In general, operations on strings are designed to be very efficient and, as a consequence, the operations may 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()** *str*.**size()**	Returns number of characters in receiver string (**length** and **size** are synonyms).
*str***[***index***]** *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* *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, default argument 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.
str.**erase**(*pos, count*)	Removes **count** chars from receiver string starting at index **pos**. If **count** is not given, default argument removes all characters to end of string. Modifies receiver string.
str1 **<** *str2* **== != < > <= >=**	The standard relational operators are overloaded to compare strings. 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

"strutils.h" adds a few conveniences for handling string conversions. These are free functions (i.e. not member functions invoked on a receiver string).

`string` **RealToString**`(double d)` `double` **StringToReal**`(string str)`	Convert double value to string form and vice versa. **StringToReal** raises an error if input is not a well-formed real value.
`string` **IntegerToString**`(int i)` `int` **StringToInteger**`(string str)`	Convert integer value to string form and vice versa. **StringToInteger** raises an error if string is not a well-formed integer value.
`string` **ConvertToUpperCase**`(string str)` `string` **ConvertToLowerCase**`(string str)`	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.

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 use **ifstream** for input file streams, and **ofstream** for output file streams. There are many other stream features than listed 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 classnames are also shortened with a typedef. The underlying full 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:

strm.**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).
strm.**close()**	Closes file. This is automatically done by stream destructor, but if you open another file on a stream, you must first explicitly close any previously open one.
strm.**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).
strm.**clear()**	Clears error state of the receiver stream, necessary to allow further operations on the stream.

These operations are specific to output streams.

ostrm **<<** *num* **<<** *str* **<<** *ch*	Stream insertion **<<** does formatted output. See **<iomanip>** for fancy format/precision/align/etc.
ostrm.**put(***ch***)**	Outputs a single char onto receiver stream.

These operations are specific to input streams.

istrm **>>** *num* **>>** *str* **>>** *ch*	Stream extraction **>>** reads formatted input. By default, skips whitespace. Puts stream into fail state if read doesn't match expected.
istrm.**peek()** *istrm*.**get()**	Read next character from receiver stream. Returns **EOF** if nothing left. Returns **int** rather than **char** in order to represent EOF. **peek** returns the character but doesn't remove it from the stream.
istrm.**unget()**	Pushes last char read back on receiver stream.
getline(istream & ist, string & str, char delimiter = '\n')	Reads next line of input (up to **delimiter**) and stores in **str** ref param. **Note:** this is a free function not a stream member function! You pass the stream to read 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 free functions.

`string` **`GetLine`**`()`
`int` **`GetInteger`**`()`
`long` **`GetLong`**`()`
`double` **`GetReal`**`()`
	Each 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 and cannot be converted to the expected type, the function reprompts and reads another line until input is valid.

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` **`Randomize`**`()`	Seeds random number generator. Should be called exactly once at start of program to establish new random sequence.
`int` **`RandomInteger`**`(int low, int high)` `double` **`RandomReal`**`(double low, double high)`	Returns int/real within given range.
`bool` **`RandomChance`**`(double probability)`	Returns true/false based on random probability.

CS106 graphics library

Our graphics library provides support for drawing in a simple graphics window. The canvas uses a Cartesian-coordinate system and units are expressed in inches (not pixels!). The base graphics library supports drawing lines and arcs. The extended graphics library adds text, filled regions, elliptical arcs, color, and support for event handling. There is a quick introduction to our graphics library in section 5.3 of the reader.

These are eight functions exported by "**graphics.h**".

`void` **`InitGraphics`**`()`	Set up empty graphics window
`void` **`MovePen`**`(double x, double y)`	Move pen to location **x**, **y**
`void` **`DrawLine`**`(double dx, double dy)`	Draw line from pen position moving **dx**, **dy**
`void` **`DrawArc`**`(double r, double start, double sweep)`	Draw arc described by **radius**, **start** angle, and **sweep** angle, starting from current pen position
`double` **`GetWindowWidth`**`()` `double` **`GetWindowHeight`**`()`	Returns graphics window size
`double` **`GetCurrentX`**`()` `double` **`GetCurrentY`**`()`	Returns current pen position

There are some 40 functions exported by **"extgraph.h"**. I listed the function declarations below, but did not attempt to repeat what is already well-commented in the header file. Please refer to **extgraph.h** for notes on how to use these functions.

`void DrawEllipticalArc(double rx, double ry, double start, double sweep)`	Elliptical arcs
`void StartFilledRegion(double density)` `void EndFilledRegion()`	Filled region support

`void DrawTextString(string text)`
`double TextStringWidth(string text)`
`void SetFont(string font)`
`string GetFont()`
`void SetPointSize(int size)`
`int GetPointSize()`
`void SetStyle(int style)`
`int GetStyle()`
`double GetFontAscent()`
`double GetFontDescent()`
`double GetFontHeight()`
	Text/font support

`double GetMouseX()`
`double GetMouseY()`
`bool MouseButtonIsDown()`
`void WaitForMouseDown()`
`void WaitForMouseUp()`
	Mouse event-handling support

`void SetPenColor(string color)`
`string GetPenColor()`
`void SetPenColorRGB(double r, double g, double b)`
`void DefineColor(string name, double r, double g, double b)`
	Color support

`void DrawNamedPicure(string name)`
`double GePictureWidth(string name)`
`double GetPictureHeight(string name)`
	Image support

`void Pause(double seconds)` `void UpdateDisplay()`	Delay, force buffered drawing onscreen
`void SaveGraphicsState()` `void RestoreraphicsState()`	Save/restore settings

`string GetWindowTitle()`
`void SetWindwoTitle(string str)`
`double GetFullScreenWidth()`
`double GetFullScreenHeight()`
`void SetWindowSize(double w, double h)`
`double GetXResolution()`
`double GetyResoution()`
	Window setup

"A good programmer is someone who looks both ways before crossing a one-way street."
- Doug Linder