# The String class

The **String** class is designed to extend GNU C++ to support string processing capabilities similar to those in languages like Awk. The class provides facilities that ought to be convenient and efficient enough to be useful replacements for **char*** based processing via the C string library (i.e., **strcpy**, **strcmp**, etc.) in many applications. Many details about String representations are described in the Representation section.

A separate **SubString** class supports substring extraction and modification operations. This is implemented in a way that user programs never directly construct or represent substrings, which are only used indirectly via String operations.

Another separate class, **Regex** is also used indirectly via String operations in support of regular expression searching, matching, and the like. The Regex class is based entirely on the GNU Emacs regex functions. See ªSyntax of Regular Expressionsº in the *GNU Emacs Manual* for a full explanation of regular expression syntax. (For implementation details, see the internal documentation in files **regex.h** and **regex.c**.)

## Constructors

Strings are initialized and assigned as in the following examples:

**String x; String y = 0; String z = "";** Set x, y, and z to the nil string. Note that either 0 or "" may always be used to refer to the nil string.

**String x = "Hello"; String y("Hello");** Set x and y to a copy of the string "Hello".

**String x = 'A'; String y('A');** Set x and y to the string value "A"

**String u = x; String v(x);** Set u and v to the same string as String x

**String u = x.at(1,4); String v(x.at(1,4));** Set u and v to the length 4 substring of x starting at position 1(counting indexes from 0).

**String x("abc", 2);** Sets x to "ab", i.e., the first 2 characters of "abc".

**String x = dec(20);** Sets x to "20". As here, Strings may be initialized or assigned the results of any **char*** function.

There are no directly accessible forms for declaring SubString variables.

The declaration **Regex r("[a-zA-Z_][a-zA-Z0-9_]*");** creates a compiled regular expression suitable for use in String operations described below. (In this case, one that matches any C++ identifier). The first argument may also be a String. Be careful in distinguishing the role of backslashes in quoted GNU C++ char* constants versus those in Regexes. For example, a Regex that matches either one or more tabs or all strings beginning with "ba" and ending with any number of occurrences of "na" could be declared as **Regex r = "\\(\t+\\)\\|\\(ba\\(na\\)*\\)"** Note that only one backslash is needed to signify the tab, but two are needed for the parenthesization and virgule, since the GNU C++ lexical analyzer decodes and strips backslashes before they are seen by Regex.

There are three additional optional arguments to the Regex constructor that are less commonly useful:

**fast (default 0)** **fast** may be set to true (1) if the Regex should be "fast-compiled". This causes an additional compilation step that is generally worthwhile if the Regex will be used many times.

**bufsize (default max(40, length of the string))** This is an estimate of the size of the internal compiled expression. Set it to a larger value if you know that the expression will require a lot of space. If you do not know, do not worry: realloc is used if necessary.

**transtable (default none == 0)** The address of a byte translation table (a char[256]) that translates each character before matching.

As a convenience, several Regexes are predefined and usable in any program. Here are their declarations from **String.h**.

```
extern Regex RXwhite;      // = "[ \n\t]+"
extern Regex RXint;        // = "-?[0-9]+"
extern Regex RXdouble;     // = "-?\\(\\([0-9]+\\.[0-9]*\\)\\|
                           //    \\([0-9]+\\)\\|
                           //    \\(\\.[0-9]+\\)\\)
                           //    \\([eE][---+]?[0-9]+\\)?"
extern Regex RXalpha;      // = "[A-Za-z]+"
extern Regex RXlowercase;  // = "[a-z]+"
extern Regex RXuppercase;  // = "[A-Z]+"
extern Regex RXalphanum;   // = "[0-9A-Za-z]+"
extern Regex RXidentifier; // = "[A-Za-z_][A-Za-z0-9_]*"
```

# Examples

Most **String** class capabilities are best shown via example.
The examples below use the following declarations.

```
String x = "Hello";
String y = "world";
String n = "123";
String z;
char*  s = ",";
String lft, mid, rgt;
```

```
Regex  r = "e[a-z]*o";
Regex  r2("/[a-z]*/");
char   c;
int    i, pos, len;
double f;
String words[10];
words[0] = "a";
words[1] = "b";
words[2] = "c";
```

# Comparing, Searching and Matching

The usual lexicographic relational operators (**==**, **!=**, **<**, **<=**, **>**, **>=**) are defined. A functional form **compare(String, String)** is also provided, as is **fcompare(String, String)**, which compares Strings without regard for upper vs. lower case.

All other matching and searching operations are based on some form of the (non-public) **match** and **search** functions. **match** and **search** differ in that **match** attempts to match only at the given starting position, while **search** starts at the position, and then proceeds left or right looking for a match. As seen in the following examples, the second optional **startpos** argument to functions using **match** and **search** specifies the starting position of the search: If non-negative, it results in a left-to-right search starting at position **startpos**, and if negative, a right-to-left search starting at position **x.length() + startpos**. In all cases, the index returned is that of the beginning of the match, or -1 if there is no match.

Three String functions serve as front ends to **search** and **match**. **index** performs a search, returning the index, **matches** performs a match, returning nonzero (actually, the length of the match) on success, and **contains** is a boolean function performing either a search or match, depending on whether an index argument is provided:

| | |
|---|---|
| **x.index("lo")** | returns the zero-based index of the leftmost occurrence of substring "lo" |

| | |
|---|---|
| | (3, in this case). The argument may be a String, SubString, char, char*, or Regex. |
| **x.index("l", 2)** | returns the index of the first of the leftmost occurrence of "l" found starting the search at position x[2], or 2 in this case. |
| **x.index("l", -1)** | returns the index of the rightmost occurrence of "l", or 3 here. |
| **x.index("l", -3)** | returns the index of the rightmost occurrence of "l" found by starting the search at the 3rd to the last position of x, returning 2 in this case. |
| **pos = r.search("leo", 3, len, 0)** | returns the index of r in the **char*** string of length 3, starting at position 0, also placing the length of the match in reference parameter len. |
| **x.contains("He")** | returns nonzero if the String x contains the substring "He". The argument may be a String, SubString, char, char*, or Regex. |
| **x.contains("el", 1)** | returns nonzero if x contains the substring "el" at position 1. As in this example, the second argument to **contains**, if present, means to match the substring only at that position, and not to search elsewhere in the string. |
| **x.contains(RXwhite);** | returns nonzero if x contains any whitespace (space, tab, or newline). Recall that **RXwhite** is a global whitespace Regex. |
| **x.matches("lo", 3)** | returns nonzero if x starting at position 3 exactly matches "lo", with no trailing characters (as it does in this example). |
| **x.matches(r)** | returns nonzero if String x as a whole matches Regex r. |

| | |
|---|---|
| **int f = x.freq("l")** | returns the number of distinct, nonoverlapping matches to the argument (2 in this case). |

# Substring extraction

Substrings may be extracted via the **at**, **before**, **through**, **from**, and **after** functions. These behave as either lvalues or rvalues.

| | |
|---|---|
| **z = x.at(2, 3)** | sets String z to be equal to the length 3 substring of String x starting at zero-based position 2, setting z to "llo" in this case. A nil String is returned if the arguments don't make sense. |
| **x.at(2, 2) = "r"** | Sets what was in positions 2 to 3 of x to "r", setting x to "Hero" in this case. As indicated here, SubString assignments may be of different lengths. |
| **x.at("He") = "je";** | x("He") is the substring of x that matches the first occurrence of it's argument. The substitution sets x to "jello". If "He" did not occur, the substring would be nil, and the assignment would have no effect. |
| **x.at("l", -1) = "i";** | replaces the rightmost occurrence of "l" with "i", setting x to "Helio". |
| **z = x.at(r)** | sets String z to the first match in x of Regex r, or "ello" in this case. A nil String is returned if there is no match. |
| **z = x.before("o")** | sets z to the part of x to the left of the first occurrence of "o", or "Hell" in this case. The argument may also be a String, SubString, or Regex. (If |

| | |
|---|---|
| | there is no match, z is set to "".) |
| **x.before("ll") = "Bri";** | sets the part of x to the left of "ll" to "Bri", setting x to "Brillo". |
| **z = x.before(2)** | sets z to the part of x to the left of x[2], or "He" in this case. |
| **z = x.after("Hel")** | sets z to the part of x to the right of "Hel", or "lo" in this case. |
| **z = x.through("el")** | sets z to the part of x up and including "el", or "Hel" in this case. |
| **z = x.from("el")** | sets z to the part of x from "el" to the end, or "ello" in this case. |
| **x.after("Hel") = "p";** | sets x to "Help"; |
| **z = x.after(3)** | sets z to the part of x to the right of x[3] or "o" in this case. |
| **z = "  ab c"; z = z.after(RXwhite)** | sets z to the part of its old string to the right of the first group of whitespace, setting z to "ab c"; Use gsub(below) to strip out multiple occurrences of whitespace or any pattern. |
| **x[0] = 'J';** | sets the first element of x to 'J'. x[i] returns a reference to the ith element of x, or triggers an error if i is out of range. |
| **common_prefix(x, "Help")** | returns the String containing the common prefix of the two Strings or "Hel" in this case. |
| **common_suffix(x, "to")** | returns the String containing the common suffix of the two Strings or "o" in this case. |

# Concatenation

**z = x + s + ' ' + y.at("w") + y.after("w") + ".";** sets z to "Hello, world."

**x += y;** sets x to "Helloworld"

**cat(x, y, z)** A faster way to say z = x + y.

**cat(z, y, x, x)** Double concatenation; A faster way to say x = z + y + x.

**y.prepend(x);** A faster way to say y = x + y.

**z = replicate(x, 3);** sets z to "HelloHelloHello".

**z = join(words, 3, "/")** sets z to the concatenation of the first 3 Strings in String array words, each separated by "/", setting z to "a/b/c" in this case. The last argument may be "" or 0, indicating no separation.

# Other manipulations

**z = "this string has five words"; i = split(z, words, 10, RXwhite);** sets up to 10 elements of String array words to the parts of z separated by whitespace, and returns the number of parts actually encountered (5 in this case). Here, words[0] = "this", words[1] = "string", etc. The last argument may be any of the usual. If there is no match, all of z ends up in words[0]. The words array is *not* dynamically created by split.

| | |
|---|---|
| **int nmatches x.gsub("l","ll")** | substitutes all original occurrences of "l" with "ll", setting x to "Hellllo". The first argument may be any of the usual, including Regex. If the second argument is "" or 0, all occurrences are deleted. gsub returns the number of matches that were replaced. |
| **z = x + y; z.del("loworl");** | deletes the leftmost occurrence of "loworl" in z, setting z to "Held". |
| **z = reverse(x)** | sets z to the reverse of x, or "olleH". |
| **z = upcase(x)** | sets z to x, with all letters set to uppercase, setting z to "HELLO" |
| **z = downcase(x)** | sets z to x, with all letters set to lowercase, setting z to "hello" |
| **z = capitalize(x)** | sets z to x, with the first letter of each word set to uppercase, and all others to lowercase, setting z to "Hello" |
| **x.reverse()**, **x.upcase()**, **x.downcase()**, **x.capitalize()** | in-place, self-modifying versions of the above. |

# Reading, Writing and Conversion

| | |
|---|---|
| **cout << x** | writes out x. |
| **cout << x.at(2, 3)** | writes out the substring "llo". |
| **cin >> x** | reads a whitespace-bounded string into x. |

| | |
|---|---|
| **x.length()** | returns the length of String x (5, in this case). |
| **s = (const char*)x** | can be used to extract the **char*** char array. This coercion is useful for sending a String as an argument to any function expecting a **const char*** argument (like **atoi**, and **File::open**). This operator must be used with care, since the conversion returns a pointer to **String** internals without copying the characters: The resulting **(char*)** is only valid until the next String operation, and you must not modify it. (The conversion is defined to return a const value so that GNU C++ will produce warning and/or error messages if changes are attempted.) |