Strings in C++

This is document was created by carefully reading the official C++ standard (ISO/IEC 14882, *Programming Languages - C++*, 1998-09-01) which has been adopted by ANSI, ISO, IEC, and ITI, and by testing everything on a seemingly reliable compiler (gnu v 2.95.4). Of course I may have made mistakes, but at least they will not be the usual mistakes that get copied blindly from one text-book or "reference" to the next. It is divided into four sections:

0. General notes
1. Ways of initialising strings ("constructors")
2. String operators (`+`, `+=`, `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `[]`)
3. String methods (special functions: everything else)
4. Converting between strings and other things.

Iterators are not covered here.

0. General Notes about Strings

0a. Internal Structure

A C++ `string` is not the same thing as a C string. In C, "string" is just a descriptive name, not really part of the language; it means an array of `char`s terminated by a zero. In C++, `string` is a defined type. A C++ string is an object which includes both an array of characters and an independent record of the length; the array of characters is not zero-terminated, and may even contain zeros within it.

0b. Access

`#include <string>` is required to access C++ strings. `<string>` and `<string.h>` are completely different things, and must not be confused.

0c. Character Encoding

The characters of a `string` are of type `char`; exactly what `char` means is permitted to vary from one implementation to another, but a `char` is practically invariably an 8 bit value. Whether the range is (-128 to +127) or (0 to +255) is also undefined, and really does vary between compilers. ASCII coding is not guaranteed, but can certainly be strongly expected.

1. Initialising and Creating strings: Constructors

1a. Default constructor, as in

```
string s;
```

Creates an empty string variable of effectively unlimited capacity, but empty. Like `s="";`

1b. Copy constructor, as in

```
   string s(t);      // where t is also a string, a char* or an array of chars
or string s=t;       // where t is also a string, a char* or an array of chars
or string s("init");
or string s="init";
```

Creates a new string which is an exact copy of `t` (or the quoted string).

1c. Substring constructor (beware! case 1d can easily be confused with this case)

```
string s(t, pos);          // where t is also a string, pos is an int
```

or `string s(t, pos, len);` // *where* `t` *is also a string,* `pos`*,* `len` *are ints*

The new string is a copy of `t`, with the first `pos` characters skipped. If `len` is supplied then only `len` characters are copied. That is, it copies the substring from position [`pos`] to position [`pos+len-1`], or to the end of the string, whichever comes first.

1d. Array of characters constructor

```
   string s(t, len);          // where t is a char* or an array of chars
or string s("init", len);     // and len is an int
```

The new string is made from the first `len` characters of `t`. The actual length of t is completely ignored: `string s("abc", 6)` still copies six characters *starting* with the `a`, `b`, `c`, and `\0`. This may cause a memory access violation if the length is wrong.

1e. Repeated Character constructor

```
   string s(len, ch);         // where len is an int, and ch is a single char
or string s(len, 'c');        // where len is an int
```

Creates a new string exactly `len` characters long, consisting of the character `ch` repeated `len` times.

2. String Operators

2a. Assignment of another string

following this declaration: `string s;`

```
s=t;          // where t is also a string.
```

Makes an exact copy of the source string `t`. `s` is a copy of `t`, not a reference (or alias), so later modifying `s` can not have any effect on `t`, or vice-versa.

2b. Assignment of an array of characters or a char-star pointer

following this declaration: `string s;`

```
s=t;          // where t is a char* or an array of chars
```

or `s="val";`

The characters of `t` are copied into `s`, up to *but not including* the first NUL `'\0'`. If `t` is not properly NUL-terminated, a memory access violation may result. The number of characters copied, and hence the resulting length of `s`, is the value that `strlen(t)` would produce.

2c. Assignment of a single character

following this declaration: `string s;`

```
s=ch;          // where ch is a char
```

or `s='c';`

`s` becomes a string of length 1, and its one character is set to `c`. This is true even if `c` is NUL, `'\0'`. `s='c'` is effectively equivalent to `s="c"`

2d. Comparisons: Relational Operators

The operators `==`, `!=`, `<`, `>`, `<=`, and `>=` may be used to compare two `string`s, a `string` with a `char*`, or a `char*` with a `string`; they *do not* perform string comparisons between two `char*`s. Remember that constant strings in quotes, `"like this"` are `char*`s, not C++ `string`s.

following this declaration: `string s, t;` the expressions

```
s==t       s!=t       s<t        s>t        s<=t        s>=t,
s==“xxx”   s!=“xxx”   s<“xxx”    s>“xxx”    s<=“xxx”    s>=“xxx”,
“xxx”==t   “xxx”!=t   “xxx”<t    “xxx”>t    “xxx”<=t    “xxx”>=t,
```

all behave in the naturally expected way.

But expressions like this: `“xxx”<“yyy”`, and `“xxx”<“yyy”` do not; they are simply pointer comparisons, they compare the addresses at which the strings are stored, not the characters in the strings.

The “naturally expected way” to compare strings is to compare character-by-character until a difference is found. If no difference is found, the strings are equal. If one string is shorter than the other, but no other differences are found, the shorter string is less than the longer one. This is normal dictionary order, and the usual character encoding is used (normally ASCII, but not so required).

There are no special operators for case-insensitive comparisons.

2e. The **+** operator: Concatenation.

The + operator joins two strings together, producing a new string that contains all of the characters of the two original ones. `“abc”+“def”` would give `“abcdef”`. For the + operator to work in this way, at least one of the operands must be properly declared as a C++ string. For example,

following these declarations: `string r, s, t;  char c;  char * xxx;`

```
r = s + t;
r = s + c;
r = c + s;
r = s + “more”;
r = “more” + s;
r = s + xxx;
r = xxx + s;
```

all perform the concatenation operation that is probably expected. But `“abc”+“def”` and `“abc”+ ‘d’` do not; they are simple C-style pointer additions.

2f. The **+=** operator: Self Concatenation.

The += operator has the same effect as + followed by =, as one might expect. The left operand must be a string, but the right one may be a string, char, or char*. `s+=t` has the same effect as `s=s+t`.

following these declarations: `string s, t;  char c;  char * xxx;`

```
s += t;
s += c;
s += “more”;
s += xxx;
```

all perform the concatenation operation that is probably expected.

2g. Indexing, **[]**.

The [] operator gives direct access to the individual characters of the string; it performs no range checking, attempting to access beyond the end of a string is an error that may go undetected.

following this declaration: `string s=“abc”;`

`s[0]` is the character `‘a’`, `s[1]` is the character `‘b’`, `s[2]` is the character `‘c’`, and `s[3]` is the character `‘\0’`; all others are undefined. The fact that `s[3]` returns `‘\0’` is due to a special requirement, it does not imply that there really is a `‘\0’` character at the end of the string.

Assigning a `char` value to `s[0]`, `s[1]`, or `s[2]` will change the string as expected; assigning to `s[3]` or later is an error that may go undetected.

3. String Methods

3a. Finding the Length

following this declaration: `string s;`
the expression `s.length()`
or `s.size()`

Returns an `int`, being the number of characters currently in the string. `length()` and `size()` are identical. This is not the capacity of the string, just the number of characters it happens to contain. The length is not computed by counting characters, as there is no end-of-string character in a C++ string; the length of a string is stored along with the characters, and does not need to be computed. The ASCII NUL character, '\0', is not significant. for example,

```
{ string s="abcde";
  s=s+'\0';
  s=s+"fghij";
  cout << s.length() << ": '" << s << "'\n";
```

will report a length of eleven, and print eleven characters between the quotes, although the sixth of those characters is invisible.

3b. Procrusteation

following this declarations: `string s;  int n;  char ch;`
`s.resize(n, ch);`
or `s.resize(n);`

Set the length of the string to exactly `n`. If `n` is less than the current length, characters are lost from the end; if `n` is more than the current length, the character `ch` (or `'\0'` if `ch` is not provided) is added repeatedly until the length reaches `n`.

3c. Clearing

following this declaration: `string s;`
`s.clear();`

Erases the contents of `s`; the length becomes zero..

3d. Checking for emptiness

following this declarations: `string s;  int n;  char ch;`
the expression `s.empty()`

is `true` if the string is empty (i.e. has zero length), and `false` otherwise. It is exactly equivalent to `(s.length()==0)`.

3e. Capacity

Strings may change size frequently. To avoid having to reallocate the memory for a string after every operation, strings are often *over-allocated*: they are given more memory than their current length actually requires. Of course, a string can't be pre-allocated an unlimited amount of memory, so when a string grows a lot reallocation is still required, but efficiency can be significantly improved by providing space before it is needed.

In nearly all cases it is useless to do so, but it is possible to find and even set the current capacity of a string. For instance, if it is known that a string will ultimately contain a million characters, but

will only reach that size gradually, it might be advantageous to set its memory allocation to one million bytes right at the beginning.

following this declarations: `string s;  int n;`
the expression `s.capacity()`

returns the current memory allocation for the string: the length it may grow to without requiring a new allocation of memory, and

`s.reserve(n);`

informs the system that the string may grow to a length of `n`, and *suggests* that it may want to allocate the extra memory that may be needed now. The suggestion may well be ignored.

and the expression `s.max_size()`

returns the maximum length that the string may every grow to, regardless of current length or current memory allocation. `max_size` can be assumed to be very big indeed, and is normally only limited by the total amount of memory in the computer if it is limited at all.

3f. Appending

The operator **+** provides the basic functionality of the `append` method in a simper form, but the full method provides more options, as described here.

following this declarations: `string s, t;  int pos, len;  char* xxx;  char c;`
the expression `s.append(t)`

is a new string which consists of the characters of `s` immediately followed by the characters of `t`; it appends `t` to the end of `s`. Appending does not modify either of the strings, it constructs a totally new one.

the expression `s.append(t, pos, len)`

is the same as the simpler version of `append`, except that it only appends part of the string `t` onto the end of `s`. The first `pos` characters of `t` are skipped, and a maximum of `len` characters from `t` are appended. `s.append("abcdefgh", 2, 3)` is the same as `s.append("cde")`.

the expressions `s.append("characters")`
and `s.append(xxx)`

append all the characters of the given C-style string to the end of `s`. The end of `xxx` is marked by the character `'\0'`.

the expressions `s.append("characters", n)`
and `s.append(xxx, n)`

append exactly `n` characters from the beginning of the given C-style string to the end of `s`. The actual length of `xxx` is ignored: `n` characters will be appended, even if a `'\0'` is encountered.

the expression `s.append(n, c)`
or `s.append(n, 'c')`

append exactly `n` copies of the character `c` to the end of `s`. The length of `s` is increased by `n`.

3g. Replacing

The operator **=** provides the basic functionality of the `assign` method in a simper form, but the full method provides more options, as described here.

following this declarations: `string s, t;  int pos, len;  char* xxx;  char c;`
the expression `s.assign(t)`

has exactly the same effect as `s=t`. The value returned is the new value of `s`. The characters of `s` are replaced with a copy of the characters of `t`: no aliassing results.

the expression `s.assign(t, pos, len)`

is the same as the simpler version of `assign`, except that it only uses part of the string `t`. The first `pos` characters of `t` are skipped, and a maximum of `len` characters from `t` are taken. `s.assign("abcdefgh", 2, 3)` is the same as `s.assign("cde")`.

the expressions `s.assign("characters")`
and `s.assignd(xxx)`

copies all the characters of the given C-style string to replace `s`. The end of `xxx` is marked by the character `'\0'`.

the expressions `s.assign("characters", n)`
and `s.assign(xxx, n)`

copy exactly `n` characters from the beginning of the given C-style string onto `s`. The actual length of `xxx` is ignored: `n` characters will be copied, even if a `'\0'` is encountered.

the expression `s.assign(n, c)`
or `s.assign(n, 'c')`

set the string `s` to a sequence of exactly `n` copies of the character `c`. The length of `s` becomes `n`.

3h. Indexing, `at()`.

The `at` method gives direct access to the individual characters of the string; it behaves exactly like the indexing operator [], except that range errors are detected.

following this declaration: `string s="abc";`

`s.at(0)` is the character `'a'`, `s.at(1)` is the character `'b'`, `s.at(2)` is the character `'c'`, Any other value will cause a run-time error: `s.at(3)` does not give `'\0'`.

Assigning a `char` value to `s.at(0)`, `s.at(1)`, or `s.at(2)` will change the string as expected; assigning to `s.at(3)` or later will cause a run-time error.