String Samples

```
#include <string>
```

Be careful, don't use `<string.h>` instead, it is a completely different thing.

Declaring a variable, function, parameter, or constant is nothing special, for example:

```
string name;
string NameOfMonth(int m);
int NumberOfSpacesIn(string s);
const string month = "February";
```

Methods

Methods are special kinds of functions that actually belong to an object, and are expected to work on that object without having been told to. For example, a normal function for finding the length of a string `s` would be called like this: `length(s)`, but a *method* for finding the length of a string would actually *be* part of that string; the function's name would include the string's name, and it wouldn't need to be given any parameter: `s.length()` returns the length of the string `s`. Strings have many pre-defined methods, some of which are useful, listed below. You can't create new methods for strings, so don't worry about how to define them. They are already there just waiting to be used. Of course, you can easily define your own normal functions that work on strings.

Other ways of initialising or setting a string:

```
string s(t, pos);
s.assign(t, pos);
```

Both of those just assign part of the string `t`, after skipping the first `pos` characters

```
string s(t, pos, len);
s.assign(t, pos, len);
```

Both of those just assign part of the string `t`, after skipping the first `pos` characters they only copy len characters of what's left.

Example:

```
string one="abcdefghijklmnopqrstuvwxyz";
string two(one, 20);
string three(one, 20, 3);
cout << "two=" << two << ", three=" << three << "\n";
two.assign(one, 10, 8);
cout << "two=" << two << ", three=" << three << "\n";
```

produces this output:

```
two=uvwxyz, three=uvw
two=klmnopqr, three=uvw
```

```
string s(num, ch);
s.assign(num, ch);
```

Both of those set the string to contain `num` copies of the character `ch`.

Example:

```
string one(5, 'x');
cout << one;
one.assign(3, 'y');
cout << one << "\n";
```

produces this output:

```
xxxxxyyy
```

Comparisons

The normal relational operators ==, !=, <, >, <=, >= may be used on strings, and behave as the naive user might expect. Usually.

Warning: For these operators to work correctly, at least one of the operands must be a proper declared C++ string, either a string variable, a const string, a string function result, or the result of a (string) typecast. Strings in `"quotes"` do not count. This: `"abc"<"xyz"` does *not* compare the two strings `"abc"` and `"xyz"`, it is a C-style pointer operation.

The comparison is performed on the individual characters of the string until the first difference is found. If no differences are found in the characters, the lengths are compared instead. The individual characters are compared using the normal encoding, which is *usually* ASCII. Comparisons are case-sensitive, capital letters are considered different from little letters. No characters in the strings, even spaces, are ignored.

Given these declarations:

```
string scat="cat", sdog="dog", sCat="Cat", sDog="Dog", sdoggie="doggie";
```

the following results should apply:

`scat<sdog`	is true, normal dictionary ordering
`scat==sCat`	is false, capital 'C' is different from little 'c'
`scat>sCat`	is true, capital 'C' comes before little 'c', in ASCII at least
`scat>sDog`	is false, capital letters before little letters, in ASCII at least
`sDog<sdoggie`	is true, capital letters before little letters, in ASCII at least
`sdog==sdoggie`	is false, they have different lengths
`sdog<sdoggie`	is true, they are the same, but the first is shorter.
`sdoggie=="dog gie"`	is false, spaces matter
`sdoggie=="doggie"`	is true.

Here is what the warning is about:

`"cat"=="cat"`	is expected to be false
`"cat"<"dog"`	has a 50-50 chance of being false (meaning that any given compiler could go either way, not that the result is likely to be different each time round a loop).

Sequences of letters inside double quotes are not C++ strings; under many circumstances they are converted into C++ strings, but not always. This is one of the places where it doesn't happen.

Appending or Enlarging Strings

`s+t`

Is a simple expression, it results in a new string that consists of all the characters of `s` immediately followed by all the characters of `t`. The length of the result is the length of `s` plus the length of `t`.

Example:

```
string one="abc", two="wxyz";
string three=one+two;
cout << "one=" << one << ", ";
cout << "two=" << two << ", ";
```

```
cout << "three=" << three << "\n";
```
produces this output, notice that one is not changed:
```
one=abc, two=wxyz, three=abcwxyz
```

```
s+=t;
s.append(t);
```
Are equivalent; they both modify the original string s by adding all the characters of t to the end of it. The resultant length of s is the original length of s plus the length of t.

```
s.append(t, pos);
```
Assuming t is an official string and pos is an integer, the string s is enlarged by adding all except the first pos characters of t onto it. Note that in this one case, t must actually be declared as a C++ string. Quoted constant "strings" behave differently.

```
s.append(cc, num);
```
Assuming cc is a quoted string of characters (e.g. "this") or a char*, and num is an integer, the string s is enlarged by adding only the first num characters of cc onto it.

Absurdity Warning
```
string one="abcdefghij", two="abcdefghij";
string extra="123456789";
one.append(extra, 4);
two.append("123456789", 4);
```
These two uses of append, which obviously should be the same, are different. A sequence of characters inside double-quotes is not a C++ string, and the automatic conversion only happens when there is no same-named function that expects a char* value. The results are that
```
one is "abcdefghij56789"
two is "abcdefghij1234"
```
As far as I am aware, this is the only such example: append with one apparent string parameter followed by exactly one int parameter.

```
s.append(t, pos, num);
```
Assuming t is a string and pos and num are integers, The string s is enlarged by adding num characters from t, after skipping the first pos of them, onto it.

Example:
```
string one="abcdefghijklmnopqrstuvwxyz";
string two="start";
two.append(one, 20);
cout << "two=" << two << "\n";
two.append(one, 3, 3);
cout << "two=" << two << "\n";
```
produces this output:
```
two=startuvwxyz
two=startuvwxyzdef
```

```
s.append(num, ch);
```
Assuming num is an integer and ch is a char, The string s is enlarged by adding num copies of the character ch onto the end of it.

Example:
```
string sss="cat";
sss.append(10, 'S');
cout << sss << "\n";
```
produces this output:
```
catSSSSSSSSSS
```

Getting information about a String

`s.length()` and `s.size()`

Are exactly the same thing. They both give the size, in characters, of the string.

Example:

```
string one="abcdefghijklmnopqrstuvwxyz";
cout << one.length() << "," << one.size() << "\n";
```

produces this output:

```
26,26
```

Accessing the Characters of a String

`s[i]`

(if `i` is an `int`, or an expression with an integer value) gives the single character at the `i`th position in string `s`. Positions are counted from zero, so `s[0]` is the first character of string `s`. If `i` is less than zero, or greater than the length of the string, this expression is unreliable: it won't work, but the error might not be detected. If `i` is equal to the length of the string, this expression will return the special character '\0'; that does not mean that there is a NUL character at the end of a string, this is just a special rule for compatibiity with C.

`s.at(i)`

(if `i` is an `int`, or an expression with an integer value) is exactly the same as `s[i]`, except that errors are properly detected: using a value of `i` that is less than zero, or greater than *or equal to* the length of the string, causes a fatal run-time error.

Example:

```
string one="abcdefghijklmnopqrstuvwxyz";
cout << one[3] << "," << one[25] << "\n";
cout << one.at(0) << "," << one.at(25) << "\n";
one[4]='*';
one.at(6)='#';
cout << one << "\n";
```

produces this output:

```
d,z
a,z
abcd*f#hijklmnopqrstuvwxyz
```

Larger examples.

This function counts up the number of spaces in any string:

```
int countspaces(string s)
{ int total=0;
  int len=s.length();
  for (int i=0; i<len; i+=1)
    if (s[i]==' ')
      total+=1;
  return total; }
```

This function replaces all the spaces in a string with dashes:

```
void changespacestodashes(string & s)
{ int len=s.length();
  for (int i=0; i<len; i+=1)
    if (s[i]==' ')
      s[i]='-'; }
```

This function converts the string to all capitals, leaving non-letters unchanged:

```
void capitalise(string & s)
{ const int difference=('a'-'A');
  int len=s.length();
  for (int i=0; i<len; i+=1)
  { char c=s[i];
    if (c>='a' && c<='z')
      s[i]=c-difference; } }
```

Searching for Substrings

In the string "A**bra**cada**bra**", the smaller string "`bra`" appears twice, as a substring. One appearance is near the beginning (with one one character before it starts), and the other is right at the end (with 8 characters before its start). The `substr` methods automate the search for substrings.

`s.find(t)`

Looks for the first appearance of `t` as a substring of `s`. If there is such a substring, its position (represented by the number of characters before it) is returned. If there is no such substring, a position outside the bounds of the string is returned instead.

`s.find(t, pos)`

(where `pos` is an integer) Looks for the first appearance of `t` as a substring of `s` after skipping the first `pos` characters. If there is such a substring, its position (represented by the number of characters before it) is returned. If there is no such substring, a position outside the bounds of the string is returned instead.

In both forms of this function, `t` may be a `string` or a single `char`.

Example:

```
string big="abracadabra", little="bra";
cout << big.find(little) << "\n";
cout << big.find(little) << "\n";
cout << big.find(little, 2) << "\n";
```

produces this output:

```
1
1
8
```

This loop finds all of the appearances of a substring. It does this by starting the next search immediately after the position found by the previous search.

```
int next=0, max=big.length()-1;
while (true)
{ int place = big.find(little, next);
  if (place<0 || place>max)
    break;
  cout << little << " found at position " << place << "\n";
  next=place+1; }
```

`s.rfind(t)`
`s.rfind(t, pos)`

Is exactly the same as the normal `find` method, except that the search starts from the end of the string, not the beginning, so if the substring appears in more than one position, it will find the last of them.

Example:

```
string big="abracadabra", little="bra";
cout << big.rfind(little) << "\n";
```

produces this output:

```
8
```

Searching for Single Characters

`s.find_first_of(t)`

Looks for the first appearance inside the string `s` of any character that appears anywhere in `t`. The value returned is an integer giving the position of that character within `s`, or a value outside the possible range if no such character is found. So, `s.find_first_of("0123456789")` finds the first digit in a string; `s.find_first_of("([{<")` finds the first appearance of any open-bracket. It is permitted for `t` to be a single `char` instead of a `string`.

Example:

```
string big="abracadabra"
cout << big.find_first_of("bra") << "\n";
cout << big.find_first_of("cd") << "\n";
```

produces this output:

```
0
4
```

because character `0` of the string "`abracadabra`" is `'a'`, which is one of `"bra"`, and character `4` of the string is `'c'`, which is one of `"cd"`.

`s.find_first_of(t, pos)`

Is the same as the simpler version of `find_first_of`, except that it skips the first `pos` characters in `s`, they are not looked at in the search.

`s.find_last_of(t)`
`s.find_last_of(t, pos)`

These methods are exactly the same as `find_first_of`, except that they scan backwards from the end of `s`. If there is more than one place in which one of the characters from `t` appears, the last place will be returned.

`s.find_first_not_of(t)`
`s.find_first_not_of(t, pos)`

These methods are exactly the same as `find_first_of`, except that instead of searching for any character that appears anywhere in `t`, they search for any character that does *not* appear anywhere in `t`. For example, `s.find_first_not_of ("0123456789")` finds the position in `s` of the first non-digit character.

`s.find_last_not_of(t)`
`s.find_last_not_of(t, pos)`

The obvious combination of `find_first_not_of` and `find_last_of`. They search backwards from the end of `s`, looking for the last character that does not appear anywhere in `t`.

Extracting Substrings

`s.substr(pos)`

(`pos` must be an integer) Creates a new `string`, which is a copy of all but the first `pos` characters of `s`. The original `s` is not modified.

Example:

```
string big=”elephants”;
string small=big.substr(4);
string middle=big.substr(5)+”y”;
cout << big << “ - “ << small <<  “ - “ << middle << “\n”;
```

produces this output:

```
elephants - hants - antsy
```

`s.substr(pos, len)`

(`pos` and `len` must be integers) Creates a new `string`, which is a copy of the first `len` characters of `s` after skipping the initial `pos` of them. The original `s` is not modified.

Example:

```
string big=”hippopotamus”;
string small=big.substr(5,3);
string middle=big.substr(8,2)+big.substr(1,4);
cout << big << “ - “ << small <<  “ - “ << middle << “\n”;
```

produces this output:

```
hippopotamus - pot - amippo
```

Inserting Substrings

`s.insert(pos, t)`

(`pos` must be an integer, `t` is another string) Modifies the string `s` by adding the characters of `t` into it; `pos` gives the insertion position: the number of characters of `s` before the insertion point.

Example:

```
string orig=“abcdefghijklmn”;
string extra=“**XYZ**”;
orig.insert(5, extra);
cout << orig << “\n”;
```

produces this output:

```
abcde**XYZ**fghijklmn
```

`s.insert(pos, num, ch)`

(`pos` and `num` must be integers, `ch` is a single `char`) Modifies the string `s` by adding the `num` copies of the character `ch` into it; `pos` gives the insertion position: the number of characters of `s` before the insertion point.

Example:

```
string orig=“abcdefghijklmn”;
orig.insert(5, 10, ‘+’);
cout << orig << “\n”;
```

produces this output:

```
abcde++++++++++fghijklmn
```

Removing Substrings

`s.erase(pos, num)`

(`pos` and `num` must be integers) Modifies the string `s` by removing `num` characters, starting with the one at position `pos`, from it. If the string is too short to have `num` characters removed, it is OK: removal just stops at the end of the string.

Example:

```
string orig="abcdefghijklmn", smaller="smaller";
orig.erase(3, 9);
smaller.erase(4, 99);
cout << orig << " - " << smaller << "\n";
```

produces this output:

```
abcmn - smal
```

Exchange

```
s.swap(t);
```

when `t` is a string, is equivalent to

```
{ string temp=s;
  s=t;
  t=temp; }
```

Pointless Operations

```
s.replace(pos, num, t);
```

when `t` is a string, is equivalent to

```
s.erase(pos, num);
s.insert(pos, t);
```

```
s.replace(pos, num, t, subpos, subnum);
```

when `t` is a string, is equivalent to

```
s.erase(pos, num);
s.insert(pos, t.substr(subpos, subnum));
```

```
s.replace(pos, num, t, numreps, ch);
```

when `ch` is a single `char`, is equivalent to

```
s.erase(pos, num);
s.insert(pos, numreps, ch);
```

Three-way Comparison

The normal comparison operators `<`, `>`, `<=`, `>=`, `==`, `!=` can be wasteful if there are three conditions to be tested for, each having its own associated action: do one thing if `a<b`, another thing if `a==b`, and a third thing if `a>b`; at least two comparisons of the same strings must be performed. A three-way comparison compares two strings, and represents the result as a numerically in a way that makes a second comparison unnecessary.

```
s.compare(t)
```

(`s` and `t` must both be strings) This is an expression with integer value. If `s<t`, the value is negative; if `s==t`, the value is zero, and if `s>t`, the value is positive.

Example:

```
string one="hello", two="cat";
int cmp = one.compare(two);
cout << "The ordering of " << one << ", " << two << " is ";
if (cmp<0)
  cout << "Correct\n";
```

```
else if (cmp==0)
  cout << "Indeterminate\n";
if (cmp>0)
  cout << "Incorrect\n";
```

"Constant Strings"

Always remember that a "constant string", a bunch of characters inside double quotes, like `"this"`, is *not* a C++ `string`. It is just an array of `char`s with a special terminator character `'\0'` added to the end. The true type is `char*`, pronounced "char-star", which is totally different from `string`.

`char* ≠ string`

`"example" ∈ char*`

`"example" ∉ string`

Whenever a function (or method) expects a `string` parameter, but a `char*` is provided instead, C++ will perform an automatic conversion. A new temporary `string` is created as needed, and passed into the function in place of the `char*` provided.

But this only applies to parameters. When trying to call a method, the attached object will not be automatically converted:

`"example".length()` does not work.

The phrase "Constant String" does not correctly describe this situation. "Constant String" is usually taken to refer to something declared as "`const string`". In the official literature a sequence of characters inside double-quotes is usually called a *String Literal*.

Compatibility with Old C

Given a C++ string, s, the char* or array of characters that it holds may be extracted. Normally this is only useful if one of the old C string library functions needs to be used, or if pure data is needed for an operating-system level call. There are two relevant methods:

`s.c_str()`
and
`s.data()`

The `c_str()` method returns a `char*` value that is completely compatible with the C programming libraries; it is a (pointer to an) array of characters, with a `0` added after the last character. If `s.length()` is `n`, then the array returned by `s.c_str()` will be at least `n+1` bytes long.

The `data()` method returns a `char*` value of exactly the same length as the original string. It does not have a `0` added after the last character.

The `char*` values returned by `s.c_str()` and `s.data()` are read-only and volatile. It is forbidden to change any of the characters in the array, and if the original string `s` is modified it becomes forbidden to even look at that array (it must be extracted again by another call).

Examples:

```
string one="hello";
printf("<<%s>>\n", one.c_str());
```

produces the output: (`#include <stdio.h>` is required)

```
<<hello>>
```

```
string one="hello";
write(1, one.data(), one.length());
```

produces the output: (`#include <unistd.h>` is required)

```
hello
```