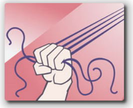
Creating *string* Objects



As in Java, **`string`** **is a library class type;** it is not part of the C++ language. As in most programming languages, the C++ **`string`** type is a sequence of characters, which can be treated as a single unit. The class is declared in the **`<string>`** header, which you **must** include, (unlike Java).

There are several different ways you to **create** **`string`**objects:

```
string s1;                  // empty string
string s2{"Hello"};         // explicitly initialized
string s3 = "World";        // Legacy C/Java style
string s4{s3};              // a copy of s3
string s5{'c', 'a', 't'};   // a sequence of chars
string s6{R"("bob")"};      // a raw string
string s7(20, '-');         // 20 dashes
```

Let's look the most useful ones.

1. In Java, **`s1`** is a *null* string. (That is, it a **`String`** variable which contains the special value **`null`**, which cannot be used. Unlike Java, in C++, it is the **empty string**.
2. **`s2`** **explicitly** converts a **string literal** (character array) to a C++ **`string`** object. String literals, such as **`"hello"`** are **not** **`string`** objects, as they are in Java. Instead, they are **pointers** to a single character at the beginning of the literal.
3. **`s3`**, the syntax you are probably most comfortable with, **implicitly** converts a C-string literal to a C++ **`string`** object.
4. Produces a **`string`** that is **a copy** of the **`string s3`**.
5. A **`string`** initialized with **a sequence** of **`char`** literals.
6. Produces a **`string`** object from a **raw string literal**. Raw string literals begin with **`R"(`** and end with **`)"`**. Inside you may store **any** character without using escape sequences.
7. Produces a **`string`** made of **`20`** **`'-'`** characters. Note that **`char`** literals use single quotes, just as they do in Java. Python does not use the **`char`** type. Note that **you must use parentheses** for this constructor, not braces.

The **`{}`** and the **`()`** may often be used interchangably. However, for **`s5`**, you **must** use the braces **`{}`**, and for **`s7`** you **must** use parentheses **`()`**. In C++98, you must use parentheses, not braces, and **`s5`** and **`s6`** will not work at all. These constructors, and raw strings were not added until C++11.

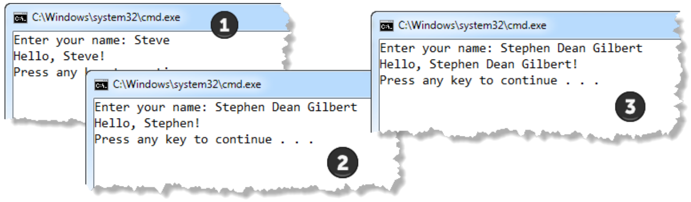
C++14 added C++ string literals, which is a regular C-string literal, with an **`s`** suffix, like **`"hello"s`**. This is no longer a pointer, but a full-fledged C++ **`string`** object, as in Java.

String Input and Output

You may use `>>` and `<<` to read and write `string` objects, like this:

```
1  cout << "Enter your name: ";
2  string name;
3  cin >> name;
4  cout << "Hello, " << name << "!" << endl;
```

This version of the program reads a **`string`** input by the user into the variable **`name`** and then includes **`name`** as part of the greeting, as shown in the screenshots below:



1. If the user enters only a first name, then all goes as you'd expect.
2. However, the user enters a **full name** instead of just the first, only the first is read.

   Even though the program contains no code to split the name apart, it somehow still uses **only** the first name when it prints its greeting.

   Why? Because **`>>` stops reading** as soon as it sees the first **whitespace character**. A whitespace character is any character that appears as blank space on the screen, and includes the tab and newline characters.

3. To read **an entire line of text,** use the **`string`** function **`getline()`** like this, in place of line 3:

   ```
   3  getline(cin, name);
   ```

   This **reads an entire line** from **`cin`** into the variable **`name`**. When run, the program allows you to display the full name of the user instead of just the first name.

Concatenation & Comparison

The `<string>` library redefines several standard operators using a C++ feature called **operator overloading**. When you use the `+` operator with numbers, it means addition, but, when you use it with the `string` type, it means **concatenation**.

```
string s1 = "hello", s2 = "world";
string s2 = s1 + " " + s2;          // "hello world"
```

The shorthand `+=` assignment operator has also been overloaded. It concatenates new text to the end of an existing `string`. You may concatenate `char` values to a `string` object, but you **cannot** concatenate numbers to `string` objectss as you could in Java.

```
string s{"abc"};  // uniform initialization
s += s;           // ok, "abcabc"
s += "def";       // literal ok, "abcabcdef"
s += 'g';         // char ok, "abcabcdefg"
s = s + 2;        // ERROR; no conversion
```

You **cannot** concatenate two string literals: `"a" + "b"` is **illegal**. However, separating them with whitespace, like `"a" "b"`, is legal. Use this is used to join long lines together.

Comparisons

C++ overloads the **relational operators** so that you can **compare** `string` values just like primitive types. To see if the value of `str` is equal to `"quit"`, just write this:

```
if (str == "quit") . . .
```

There is no need to use `equals()` or `compareTo()` as in Java.

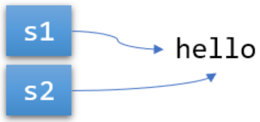
Strings are compared using **lexicographic ordering**. Informally that means a `string` is smaller if it would appear earlier in the dictionary. However, when doing comparisons, case is significant, so `"abc"` is **not** equal to `"ABC"`. Upper-case characters are "smaller" than lower-case characters, because they have smaller ASCII values.

Mutability & Value Assignment

In C++ `string` objects are mutable; you may change the individual characters inside a `string` variable. Compare this with Java or Python, where string objects are immutable.

```
string str = "hello";
str[0] = 'j';
cout << str << endl;    // prints jello
```

In Java and in Python, assignment of object types means that the variables are copied, but that the objects are not. Here' a piece of Java code which creates a `String s1` and then creates a second, `s2` initialized with `s1`. The illustration shows what this looks like in memory.




```
String s1 = "hello";
String s2 = s1;
```

C++ works differently. In Java and Python, variables refer to objects; in C++ variables contain objects. In C++, assigning one `string` to another, **copies the underlying characters** into an entirely new `string`, in the same way that assigning one `int` variable to another creates a new, independent variable and value.

Languages (like C++) that work like this have **value semantics**. In C++, the statement

```
str1 = str2
```

overwrites any previous contents of `str1` with a **copy** of the characters contained in `str2`. The variables `str1` and `str2` therefore remain **independent**, which means that changing the characters in `str1` does not affect `str2`.

Member Functions

Because `string` is a library or class type, it also has methods, just like the Java **`String`** class has methods such as **`length()`**, **`toUpper()`** and **`charAt()`**. In C++ instead of calling these **methods**, we use the term **member function** instead. Let's look at the difference between a regular (or "free") function in C++, and a member function.

In the **`string`** class, you've already seen the **`getline()`** function. The prototype for **`getline()`** looks like this:

```
istream& getline(istream&> in, string& str);
```

The function has **two parameters**: the input stream to read from, and the **`string`** object to modify; it returns a reference to its input stream (which may be ignored).

```
string line;
getline(cin, line);
```

Although **`getline()`** is declared inside the **`<string>`** header, it **is not** part of the **`string class`**; it is just a regular function. **Member functions**, in contrast, **are part of a class**, and, as in Java, they are called by using a special syntax:

```
receiver.request(arguments);
```

In this case, *receiver* is an **object**, and *request* is a member function defined in that class. When compiled, the address of the *receiver* object is passed to the member function as an **invisible** or **implicit** first parameter. Inside the member function, that implicit parameter is accessed using the keyword **`this`**, in a manner similar to Java.

String Members

Below are the member functions you should memorize:

String members	
size	the number of characters in the string (may also use length)
empty	true if the string contains no characters
at	an individual character at a particular position (may also use [ ])
front, back	the character at the front, and at the back (C++11)
substr	a new string created from a portion of an existing string
find, rfind	index of the substring searched for (from front or back)

You can look up the rest.

The *size* Member Function

`s.size()` returns the **number of characters** in the `string s`. For historical reasons, you can also use `length()`, but all of the other collections in the library use `size()`, so you should probably get used to using that. (Plus, it's less typing 😄).

The `size()` member function returns an **unsigned integer**, not an `int` as it does in Java, which may be defined differently on different platforms.

- On an embedded platform, with little memory, `size()` could return a 16-bit `unsigned short`.
- More commonly, strings can be as big as 4 billion characters, so an `unsigned int` is often large enough.
- However, you can't assume that is true. I recently recompiled some older code and discovered several places where I had assumed that `size()` returned an `unsigned int`, but the platform I was on used a 64-bit `unsigned long` instead.

This seems complex, since you don't want to re-edit your code each time you move to a new compiler. Here are three different ways to store the value returned from calling `size()` that work regardless of the platform:

```
string str{...};  // string of any size
string::size_type len1 = str.size();
auto len2 = str.size();
size_t len3 = str.size();
```

1. To be slavishly, pedantically correct, use `string::size_type`.
2. Use `auto` which **infers** the type from the initializer. (You must use `=`, not braces.)
3. Use the type `size_t`. This is the `unsigned` machine type, so your code will be adjusted automatically for each platform.

I believe that the easiest method is the last, and that's what I'll do in this class.

Characters

Individual characters in C++ are represented by the built-in primitive data type named `char` (usually pronounced "tchar", not "kar"). In memory, these values are represented by assigning each character an 8-bit integer code called an ASCII code . (Actually, only 7-bits are defined by C++, so the ASCII values 128-255 are non-standard and may vary from platform to platform.)

You write **character literals** by enclosing each character **in single quotes**. Thus, the literal `'A'` represents the internal code of the uppercase letter **A**.

In addition, C++ allows you to write **special characters** in a multi-character form beginning with a back-slash (`\`). This form is called **an escape sequence**. This includes the **newline** (`\n`), the **tab** (`\t`), and a double-quote inside a string literal (`\"`). Here is a list of the C++ escape sequences .

Character Functions

It is useful to have tools for working with individual characters. The `<cctype>` header contains a variety of functions that do that. There are two kinds of functions.

- **Predicate classification functions** test whether a character belongs to a particular category. Calling `isdigit(ch)` returns true if `ch` is one of the digit characters in the range between `'0'` and `'9'`. Similarly, `isspace(ch)` returns true if `ch` is any of the characters that appear as white space on a display screen, such as spaces and tabs.
- **Conversion macros** make it easy to convert between uppercase and lowercase letters. Calling `toupper('a')`, for example, returns the character `'A'`. If the argument is **not** a letter, the function returns it unchanged, so that `tolower('7')` returns `'7'`.

Selecting Characters

Positions in a string are subscripted (or indexed) starting at `0`. The characters in the `string` "*hello, world*" are index like this:

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

The numbers are alled the **index** or **subscript**; they must be positive (unlike Python where subscripts can be negative). Indexes start at `0` because it represents how many steps you need to travel from the beginning of the `string` to get to the element you are interested in. To retrieve the `'e'`, you have to travel one character from the beginning, so its subscript is `1`.

The <string> library has four ways to select characters from a non-empty string:

- Use the **subscript operator** like this: `cout << str[0];`
- Use the **member function** `at()` like this: `cout << str.at(0);`
- Use the members `front()` and `back()` in C++ 11+: `cout << str.front();`

If the `string` variable `str` contains `"hello, world"`, all of these expressions refer to the character `'h'` at the beginning of the `string`.

The `at()` member function makes sure the index is **in range**; the subscript operator does not. Using the subscript operator when a subscript is out of range **is undefined**. You should generally use `at()` unless you are certain that your indexes are in range.

Selecting an individual character in a `string` returns a **reference to the character** in the `string` **instead of a copy of that character**, as Java's `charAt(index)` method does. You **may assign a new value** to that reference, like this:

```
str[0] = 'H';       // or
str.at(0) = 'H';    // works as well
```

Both lines change the value from `"hello, world"` to `"Hello, world"`.

Substrings

To create a new string, initialized with only a portion of an existing string (called **a substring**), use the member function named `substr()` which takes two parameters:

- the index of the **first character** you want to select
- the desired **number of characters**.

Calling `str.substr(start, n)` creates **a new `string`** by extracting `n` characters from `str` starting at the index position specified by `start`. For example, if `str` contains the `string` `"hello, world"`, then the following code prints `"ell"`.

```
string str{"hello, world"};
cout << str.substr(1, 3) << endl;
```

The `string` begins at `0`, so the character at index `1` is the character `'e'`.

Be careful with the `substr()` function, when switching between Java and C++. In Java, the second parameter to its `substring()` method is the ending index; in C++, though, it is the number of characters in the returned substring. Forgetting this can lead to hard-to-find bugs (and crashes).

The second argument in `substr()` **optional**; if missing, `substr()` returns the substring that starts at the index and continues to the end. For instance,

```
cout << str.substr(7) << endl;
```

returns the `string` `"world"`. While this line

```
cout << str.substr(str.size() / 2) << endl;
```

uses `substr()` to print the second half of `str`, which includes the middle character if the size of `str` is odd:

*When using the `substr(start, end)` version of `substr()`, if `n` is supplied but fewer than `n` characters follow the starting position, `substr()` returns characters only up to the end of the original `string`, instead of causing a runtime error. If, however, `start` is beyond the length of the `string`, **you will** get an error. If `start` is equal to the length of the `string`, then `substr()` returns the empty string.*

Searching a String

To search for both characters and substrings, the `string` class contains a member function `find()`, which comes in several forms. The simplest form looks like this:

```
auto index = str.find(target)
```

The argument `target` is what you're looking for.

- `target` may be a `string`, a `char` or a C-string **literal**.
- The function searches through `str` looking for the **first occurrence** of `target`.
- If `target` **is** found, `find()` returns the index at which the match begins. Use `auto` or `size_t` to store this.

If you want to find the **last occurrence** of target, use `rfind()` instead.

Not Found

If `target` **is not found**, then `find()` returns the constant named `string::npos`. This constant is defined as part of the string class and therefore requires the `string::` qualifier. This is a good candidate for a **named constant** in your code:

```
const auto kNotFound = string::npos;
```

The `find()` member function takes an optional second argument to indicate the index at which to start the search. Both styles of the `find()` member function are illustrated here:

```
string str{"hello, world"};
auto a = str.find('o');         // char, 4
auto b = str.rfind("o");        // C-string, 8
auto c = str.find('l', 4);      // 10
auto d = str.find("waldo");     // string::npos
```

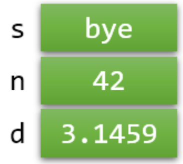
The `find()` member functions consider uppercase and lowercase characters to be different. Unlike Java, there is no built-in `toUpperCase()` or `toLowerCase()` member function in the `string` class.

Variations

In addition to `find()` and `rfind()`, you can find the position of the first (or last) occurrence of a character that **appears in a set** or that **doesn't** appear in a set. Here are some examples:

```
string s{"\"Hooray\", the crowd cheered!"};
auto a = s.find_first_of("aeiou");      // first lower-case vowel
auto b = s.find_last_of("\",.!:;");     // last punctuation
auto c = s.find_first_not_of(" \t\n");  // first non-whitespace
```

References



In C++, both library types, like `string`, and the built-in primitive types, like `int` and `double`, are called **value types**. In C++ such variables are "boxes" that **contain** data.

C++ also has several **derived types**:

- **pointers**, which contain the address of a variable,
- **arrays**, which contain a sequence of variables
- **references**, which provide an **alias** or alternate name for an existing variable

A reference name is an alternate name or **alias** for an existing object. Here's an example of a variable `n` and its alias `r`. You create a reference by putting an ampersand (`&`) after the type name. The type of `r` is usually pronounced *"int-ref"*.

```
int n = 3;
int& r = n;     // r is now an alias for n
r = 42;         // n is also now 42
```

Here, `r` is simply an alternate name for `n`. It **is not** a new variable. Under the hood, the compiler often uses pointers to implement references (although that's not required). Even if you understand how pointers work, however, you should try not to get the two concepts confused.

Conversions & Const-ref

Unlike value-type variables, references have no implicit conversions. For instance, the following compiles and runs, because even though `a` is type `int` and `b` is type `double`, the compiler will **implicitly** create a temporary `int` value to "stand in" for `b`.

```
int a = 42;
double b = a;     // implicitly double b = int(a)
```

The following code, however, **will not** compile, because `x` is an `int`, but `rx` is a **reference to** a `double`. If `rx` were a `double`, (as in the previous example), instead of a `double&` then `x` **would be** promoted and stored in `rx`.

```
int x = 3;
double& rx = x;     // ILLEGAL. x is not a double
```

Constant References

While regular references must refer to an ***lvalue*** of exactly the same type, a **constant reference** may refer to a **literal** or **temporary** value. Here are some examples:

```
int& lit = 3;                // ILLEGAL
const int& lit2 = 3;         // OK, const-ref
string& str = "OK";          // ILLEGAL
const string& str2 = "OK";  // OK
```

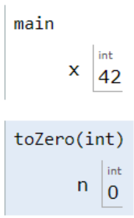
Reference Parameters

When you pass a variable to a function, the function receives a copy of the calling value or **argument**. Assigning to a parameter variable changes the parameter but has no effect on the argument. Consider this program, along with a function which attempts set a variable to zero:

```
void toZero(int n) { n = 0; }

int main()
{
    int x = 42;
    toZero(x);
    cout << x << endl;
};
```

If you call the procedure the parameter variable named `n` is initialized **with a copy** of the value is stored in `x` (`42` in this case). Making a copy of arguments when calling a function, is known as **pass by value** or **call by value**, and the parameter `n` is known as a **value parameter**.



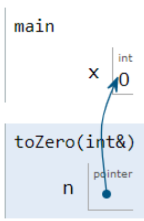
The assignment statement `n = 0;` inside the function sets **the parameter variable** `n` to `0` but leaves the variable `x` unchanged in the `main` function.

Pass by Reference

If you want to change the value of the calling argument, you can change the parameter from a value parameter into a **reference parameter** by adding an ampersand between the type and the name in the function header, like this:

```
void toZero(int& n) { n = 0; }
```

Unlike value parameters, **reference parameters are not copied.** Instead, the function treats `n` as **a reference to the original variable**, which means that the memory used for that variable is shared between the function and its caller.



If you trace through the program by clicking the link, you'll see that this time, the variable `x` in `main` is set to `0`, just as you intended.

String Value Parameters

Imagine you want to write a function named `count_vowels()`, which counts the number of vowels in a **`string`**. Here's a first attempt:

```
int count_vowels(string str) {
  int vowels = 0;
  for (char c : str) {
    switch (c) {
      case 'a': case 'A': case 'e': case 'E': case 'i':
      case 'I': case 'o': case 'O': case 'u': case 'U':
      vowels++;
    }
  }
  return vowels;
}
```

The code in this function is correct, readable, and quite efficient. However, it has **one flaw**. Imagine calling the function with a long **`string`**, say the text of *War and Peace*. Because the parameter variable **`str`** is a **value** parameter, your code will make a copy of the whole text of the book and store that in **`str`**.

string book =  ;
int vowels = count_vowels(book);
int count_vowels(string  str)

Thus, using pass-by-value with **`string`** arguments is **very inefficient**.

> *Never pass class types, such as **`string`** and **`vector`** by value.*

String Reference Parameters

Since reference parameters don't make a copy of the argument, they are much more efficient when passing a class-type argument such as **`string`** or **`vector`**. What if you were to change the heading of **`count_vowels`** like this. Would that work?

```
int count_vowels(string& str)
```

Well, yes and no!

- Because the parameter **`str`** is now a **reference**, there is no copy made, so it is **much more efficient**.
- However, because it is a reference, you can now only call the **`count_vowels`** function with an ***lvalue***. You could no longer write: `count_vowels("hello");`. Your function is much less **usable**.
- Finally, since **`str`** is a reference, there is nothing to **prevent** the **`count_vowels`** function from **inadvertently modifying** the parameter, and, thus by extension, the argument. The function is not as **safe** as it could be.

The solution is simple, however. **Whenever** you pass a **`string`** as an argument to a function, use **`const string&`** for the parameter if the function **will not** modify the calling argument, and **`string&`** if it will.

Here is the improved header for **`count_vowels`**, which is correct, efficient and safe.

```
int count_vowels(const string& str)
```

*If the **`string`** **should** be modified use a regular reference. If the string **should not** be modified, use a **`const`** reference as your parameter type.*

You can add these C++11 **type alias declarations** to your programs to make this easier if you like:

```
using stringIn = const string&;   // input string not modified
using stringRef = string&         // output string, modified
```