Firefox Download Button

Strings and String Manipulation in C++

Carlos Moreno

A picture may be more misleading than a thousand words
– Carlos Moreno

C++ provides convenient and powerful tools to manipulate strings. This tutorial shows some of the basic string manipulation facilities, with examples to illustrate their use. It also shows some extensions the C++'s string capabilities by making use of some of the Boost Library facilities.

Strings and Basic String Operations

Putting aside any string-related facilities inherited from C, in C++, strings are not a built-in data type, but rather a Standard Library facility. Thus, whenever we want to use strings or string manipulation tools, we must provide the appropriate **`#include`** directive, as shown below:

```
#include <string>
using namespace std;    // Or using std::string;
```

We now use **`string`** in a similar way as built-in data types, as shown in the example below, declaring a variable **`name`**:

```
string name;
```

Unlike built-in data types (**`int`**, **`double`**, etc.), when we declare a **`string`** variable without initialization (as in the example above), we *do have* the guarantee that the variable will be initialized to an empty string — a string containing zero characters.

C++ strings allow you to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cout << "Enter your name: " << flush;
cin >> name;
    // read string until the next separator
    // (space, newline, tab)

    // Or, alternatively:
getline (cin, name);
    // read a whole line into the string name

if (name == "")
{
    cout << "You entered an empty string, "
         << "assigning default\n";
    name = "John";
}
```

```
else
{
    cout << "Thank you, " << name
         << "for running this simple program!"
         << endl;
}
```

C++ strings also provide many string manipulation facilities. The simplest string manipulation that we commonly use is concatenation, or addition of strings. In C++, we can use the **+** operator to concatenate (or “add”) two strings, as shown below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
    // result now contains "hello world"
```

Notice that both `s1` and `s2` remain unchanged! The operation reads the values and produces a result corresponding to the concatenated strings, but doesn't modify any of the two original strings.

The **+=** operator can also be used. In that case, one string is appended to another one:

```
string result;
string s1 = "hello";
    // without the extra space at the end
string s2 = "world";
result = s1;
result += ' ';
    // append a space at the end
result += s2;
```

After execution of the above fragment, `result` contains the string `"hello world"`.

You can also use two or more **+** operators to concatenate several (more than 2) strings. The example below shows how to create a string that contains the full name from first name and last name (e.g., `firstname = "John"`, `lastname = "Smith"`, `fullname = "Smith, John"`).

```
string firstname, lastname, fullname;

cout << "First name: ";
getline (cin, firstname);
cout << "Last name: ";
getline (cin, lastname);

fullname = lastname + ", " + firstname;
cout << "Fullname: " << fullname << endl;
```

Of course, we didn't need to do that; we could have printed it with several `<<` operators to concatenate to the output. The example intends to illustrate the use of strings concatenation in situations where you need to store the result, as opposed to simply print it.

Now, let's review this example to have the full name in format `"SMITH, John"`. Since we can only convert characters to upper case, and not strings, we have to handle the string one character at a time. To do that, we use the square brackets, as if we were dealing with an array of characters, or a vector of characters.

For example, we could convert the first character of a string to upper case with the following code:

```
str[0] = toupper (str[0]);
```

The function **toupper** is a Standard Library facility related to character processing; this means that when using it, we have to include the **<cctype>** library header:

```
#include <cctype>
```

If we want to change all of them, we would need to know the length of the string. To this end, strings have a method **length**, that tells us the length of the string (how many characters the string has).

Thus, we could use that method to control a loop that allows us to convert all the characters to upper case:

```
for (string::size_type i = 0; i < str.length(); i++)
{
    str[i] = toupper (str[i]);
}
```

Notice that the subscripts for the individual characters of a string start at zero, and go from **0** to **length-1**.

Notice also the data type for the subscript, **string::size_type**; it is recommended that you always use this data type, provided by the **string** class, and adapted to the particular platform. All string facilities use this data type to represent positions and lengths when dealing with strings.

The example of the full name is slightly different from the one shown above, since we only want to change the first portion, corresponding to the last name, and we don't want to change the string that holds the last name — only the portion of the full name corresponding to the last name. Thus, we could do the following:

```
fullname = lastname + ", " + firstname;

for (string::size_type i = 0; i < lastname.length(); i++)
{
    fullname[i] = toupper (fullname[i]);
}
```

Search Facilities

Another useful tool when working with strings is the **find** method. This can be used to find the position of a character in a string, or the position of a substring. For example, we could find the position of the first space in a string as follows:

```
position = str.find (' ');
```

If the string does not contain any space characters, the result of the find method will be the value **`string::npos`**. The example below illustrates the use of **`string::npos`** combined with the **`find`** method:

```
if (str.find (' ') != string::npos)
{
    cout  << "Contains at least one space!" << endl;
}
else
{
    cout  << "Does not contain any spaces!" << endl;
}
```

The **`find`** methods returns the position of the *first* occurence of the given character (or **`string::npos`**). We also have the related **`rfind`** method — the **`r`** stands for *reverse* search; in other words, **`rfind`** returns the position of the last occurence of the given character, or **`string::npos`**. You could also look at it as the first occurence while starting the search at the end of the string and moving backwards.

The **`find`** and **`rfind`** methods can also be used to find a substring; the following fragment of code can be used to determine if the word **`"the"`** is contained in a given string:

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

if (text.find ("the") != string::npos)
{
    // ...
```

For both cases (searching for a single character or searching for a substring), you can specify a starting position for the search; in that case, the **`find`** method will tell the position of the first occurrence of the search string or character after the position indicated; the **`rfind`** method will return the position of the last occurence before the position inticated — you could look at it as the first occurence while moving backwards starting at the specified position. The requirement for this optional parameter is that it must indicate a valid position within the string, which means that the value must be between **`0`** and **`length-1`**

Brainteaser: there is an additional condition related to the optional starting position when using **`find`** or **`rfind`**; can you see it?

The following fragment of code shows how to test if a string contains at least two spaces. It performs one search for a space, and then it does a second search, starting at the position where the first one was found (actually, one element after — can you see why?):

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

string::size_type position = text.find (' ');
if (position != string::npos)
{
    if  (text.find (' ', position+1) != string::npos)
    {
        cout << "Contains at least two spaces!" << endl;
    }
    else
    {
        cout << "Contains less than two spaces!" << endl;
    }
}
else
{
    cout << "Contains no spaces!" << endl;
}
```

Brainteaser: there is a bug in the fragment of code above — can you see it? even more fun: can you fix it?

There are several other string facilities that are related to `find`. Two of them are `find_first_of`, and `find_first_not_of`. Instead of finding the first occurrence of an exact string (as `find` does), `find_first_of` finds the first occurrence of any of the characters included in a specified string, and `find_first_not_of` finds the first occurrence of a character that is not any of the characters included in the specified string. An example of use is shown below:

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

if (text.find_first_of ("aeiouAEIOU") == string::npos)
{
    cout << "The text entered does not contain vowels!"
         << endl;
}
```

In this case the expression `text.find_first_of ("aeiouAEIOU")` returns the position of the first occurrence of any of the characters included in the string passed as parameter — a string containing all the vowels. `find_first_of`, like `find`, returns the position of the found character, or `string::npos` if no character matching the specified requirement was found. In this example, if `find_first_of` returns `string::npos`, that means that the string did not contain any vowel (lowercase or capital).

`find_first_not_of` works in a similar way, except that it finds the first character that is not one of the characters specified in the string, as shown in the example shown below:

```
string card_number;
cout << "Enter Credit Card Number: ";
getline (cin, card_number);

if (card_number.find_first_not_of ("1234567890- ") != string::npos)
{
    cout << "The card number entered contains invalid characters"
         << endl;
}
```

Like `find`, both `find_first_of` and `find_first_not_of` accept also an extra parameter indicating the position where to start the search (naturally, if the parameter is omitted, the search starts at position `0`).

Guess what? We also have the methods `find_last_of` and `find_last_not_of`. I'm certain that you will easily figure out how they work, and with your vivid imaginations will come up with examples of use.

Regular Expressions with the Boost Library

Notes:

1. This section assumes that you are familiar with regular expressions (at least at an intermediate level); I provide *very brief* explanations of the regular expressions used, mostly as a reminder; if you are unfamiliar with these, I could recommend this site, or Wikipedia's page on regular expressions.
2. Regular expressions are going to be part of the C++ language (part of the Standard Library), in the upcoming revision of the language, the so-called “C++0x”; however, this is still not the case (compilers provide *experimental* support for this and other new features in C++0x), so in this tutorial, I stick to the solid Boost library facilities. Still, C++ native regular expression facilities are very similar to (since they're based on) Boost's facilities, so what you learn from this tutorial will be useful when C++0x becomes official and compilers start supporting native regular expression facilities.

Regular expressions provide a mechanism for pattern matching between two strings; a regular expression is a fragment of text (a string) defining a pattern that describes a family of fragments of text. This pattern may be used to determine if another string matches it (i.e., if the other string belongs to the family of strings described by the regular expression).

This is a much more flexible and powerful tool than the basic search facilities that I presented in the previous section. A pattern with several components (or several conditions) can be specified in a single operation, providing extra flexibility.

A simple example of pattern matching is the following: suppose that a program asks the user to enter their fullname. The user could enter it in the form of first name followed by last name (e.g., `"Carlos Moreno"`), or in the form of last name, followed by comma, followed by first name (e.g., `"Moreno, Carlos"`).

You want your program to determine which form the user entered the full name in, under the assumption that if there is no comma, then it is first name followed by last name. This is very simple to do with `find`, but I will use it for this example, since it illustrates quite nicely the use of regular expressions.

Regular expressions imply pattern matching, so we express the problem in terms of a pattern for which we want to determine if the given string (the name that the user entered) matches it.

The pattern in question is very simple: `".*, .*"`. In a regular expression, the character `*` means *the preceding item any number of times* (including possibly zero times). The preceding item in this case is a dot, which means *any character*. So, the pattern above is matched by any string that contains a sequence of any number of characters, followed by comma, followed by a space, followed by a sequence of any number of characters. In the case of my name, entered as `"Moreno, Carlos"`, the string clearly matches the pattern.

This simple pattern matching based on the regular expression `".*, .*"` is not very good for this particular purpose. We notice that there are many other examples of strings that match the pattern, even though they are not really intended to pass the validation: `", xxx"` — a sequence of zero characters, comma-space, a sequence of three `x` characters; `", "` — a sequence of zero characters, comma-space, a sequence of zero characters; `",,,,,    "` — a sequence of four commas, comma-space, a sequence of three space characters; `"Aa, Bb, Cc, "` — this one matches in three possible ways (right?).

The approach of a simple `name.find (", ") != string::npos` exhibits the same problem — all those strings pass the validation even though they're not supposed to. Making a sophisticated validation with the basic C++ string facilities would be quite tough. With regular expressions, however, it is as simple as coming up with a more precise regular expression describing more specifically the pattern that we want: `[A-Z]+[a-z]*, [A-Z][a-z]*`.

The program below uses the Boost Library regular expression facilities to validate a full name with the regular expression given above. The color highlight indicates the lines that are directly related to the Boost Library regular expressions facilities.

```
#include <iostream>
#include <string>
using namespace std;

#include <boost/regex.hpp>

int main ()
{
    boost::regex fullname_regex ("[A-Z]+[a-z]*, [A-Z][a-z]*");

    string name;
    cout << "Enter you full name: " << flush;

    getline (cin, name);
    if (! regex_match (name, fullname_regex))
    {
        cout << "Error: name not entered correctly" << endl;
    }

    return 0;
}
```

In the above program, the first statement declares an object (a variable) that represents the regular expression and all the related functionality — we initialize it with the string containing the regular expression between the round brackets. In this particular example, the variable is called **`fullname_regex`**, indicating that it is the regular expression for fullname validation, and its data type is **`boost::regex`**.

The validation itself is done with the function **`regex_match`** — this function receives two parameters: the string to validate, and the regular expression object. It returns a boolean that can be used directly as a condition — **`true`** if the given string matches the regular expression, **`false`** otherwise.

When compiling this program, you have to instruct the compiler to *link* the Boost library together with the executable; this is due to the fact that we are using facilities that are not part of the C++ language, so the compiler needs to be specifically instructed on how to handle it.

On a Unix/Linux system, many of which come with the Boost library included as part of the distribution, you would do something like this:

```
c++ -o regex_test regex_test.c++ -lboost_regex
```

Depending on your system and the way the Boost library was installed, the details may vary. Before using it, you should check the documentation corresponding to the Boost library for your particular system configuration.

Substrings

We can extract one portion of a string with the method **`substr`**. This does not remove the portion from the original string; instead, it creates a new string that contains the specified portion of the original string. The required substring is specified by the starting position and the number of characters, taking into account that the position of the first character in the string is **`0`**. The following example assigns the variable fragment with the sequence of 5 characters starting at position 6 of the variable text:

```
string text = "hello world, this is a test";
string fragment = text.substr (6, 5);
```

```
    // start at 6, take 5 characters
```

The variable fragment will contain the string **"world"**, and the variable text remains unchanged.

If we omit the second parameter, then **substr** will return a substring starting at the specified position and taking all of the characters after that one. For example:

```
string text = "hello world";
string subs = text.substr (3);
```

assigns the string **"lo world"** into the variable subs.

The first parameter must indicate a valid position within the string (similar to the case of the starting position for **find**, discussed earlier), or the behaviour is undefined. The second parameter, combined with the first one, can be such that the length of the string is exceeded. In such case, the returned substring will contain as many characters as possible (it will take all the characters from the specified starting position to the end of the string), as shown in the following fragment:

```
string text = "012345";
string fragment = text.substr (2,3);    // ok

fragment = text.substr (6,2);           // Not ok (1)

fragment = text.substr (3,10);          // ok (2)
```

(1) This statement invokes undefined behaviour, since the starting position falls outside the string.

(2) Returns all the available characters starting at 3. In this particular case, the statement has the same effect as **fragment = text.substr (3,3)**.

Erasing and Replacing Substrings

In addition to extracting substrings, as we saw in the previous section, we can also modify a given string by manipulating a fragment of it; in particular, we can replace a given substring with another string, or erase the substring, causing the original string to “shrink”.

The **erase** method receives two parameters, specifying a substring (that is, starting position and number of characters) to be removed from the string, as shown below:

```
string text = "This is a test";
text.erase (5,5);
```

After executing the statement **text.erase(5,5)**, the string contains **"This test"** — the substring starting at position 5 (the **i** in **is**) and taking 5 characters (up to and including the space after the **a**) is removed from the string, causing it to shrink (you may find a good analogy with using the *Delete* key in a text editor or word processor).

Instead of removing the substring, we could replace it with another string, as shown in the example below:

```
string text = "This is a test";
```

```
text.replace (5,2,"was");
```

After executing the statement **`text.replace(5,2,"was")`**, the string **`text`** contains **`"This was a test"`**. Notice that we specified a substring of 2 characters and replaced it with a string of 3 characters. This is perfectly valid; the substring can be replaced with one of any length — if shorter, the original string will shrink; if longer, the original string will “expand” in length (similar to typing while in “insert mode” in a text editor or word processor); and if equal, the length of the string remains the same, and the substring is replaced letter-by-letter (similar to typing while in “overwrite mode” in a text editor or word processor)

Assembling Strings with String Streams

In the previous sections, I presented some of the basic facilities, including operators to concatenate strings. These facilities can be used to “assemble” a string piece by piece. However, they do not allow you to combine pieces of different data types. You can only concatenate strings with other strings, or with characters.

String streams provide an additional level of flexibility — they provide you with all the functionality of streams (e.g., **`cout`**) to assemble a string the same way you would “assemble” the output that you send to the console using **`cout`**.

Their use is quite straightforward: just keep in mind what you do with **`cout`**, plus a couple minor details. The example below builds a string that contains the following:

```
Name: Lastname, Firstname. Birthdate: YYYY-MM-DD, Age: XX
```

Assuming that we have the various pieces in variables **`firstname`**, **`lastname`** (strings), **`year`**, **`month`**, **`day`**, and **`current_year`** (integers).

```
#include <sstream>
    // ...

ostringstream person_info;

person_info << "Name: "
            << lastname << ", " << firstname << ". ";
person_info << "Birthdate: "
            << year << '-' << month << '-' << day << ", "
            << "Age: " << current_year - year;

    //  At this point, person_info.str() provides the result.
    //  You could do, for instance:

cout << "Person info:\n" << person_info.str() << endl;
```

Notice that the use of string streams requires that you provide an **`#include <sstream>`** directive.

The output produced by the above fragment of code would be something like:

```
Person info:
Name: Lastname, Firstname. Birthdate: YYYY-MM-DD, Age: XX
```

Of course, if we only had to output this information to the console, we could have done it all directly with **`cout`**. We use an **`ostringstream`** when we need to store the concatenated result in a string variable — in the example above, the string is given by **`person_info.str()`**, and I simply used that string to print it with **cout** to keep the example as simple as possible.

In cases where we need to reuse an **ostringstream**, we can “reset” it (clear its internal contents) to start over by passing an empty string to the **str** method, as shown below:

```
#include <sstream>
// ...

ostringstream out;
out << ··· ;
// Done --- do something with out.str() ...

out.str("");    // Reset it
out << ··· ;
```

Parsing Strings with String Streams

As much as we can “assemble” strings from several pieces using **ostringstream**, we can do the opposite process, breaking up a string into several pieces, using **istringstream** (the "o" stands for *output* — as in, we *output* some data to a string; the "i", naturally, stands for *input* — as in, we *input* information from a string).

And not surprisingly, as much as **ostringstream** is used in a way similar to the way we use **cout**, we use **istringstream** in a way that is very similar to the way we use **cin**. The example below shows this by taking a string with a date in ISO format (YYYY-MM-DD) and extracting the values of year, month, day into three integer variables:

```
#include <sstream>
// ...

string iso_date;
int year, month, day;
char dash1, dash2;    // Dummy variables to read delimiters

cout << "Enter date (yyyy-mm-dd): " << flush;
getline (cin, iso_date);

istringstream (iso_date);    // load value into string stream
iso_date >> year >> dash1 >> month >> dash2 >> day;
```

Again, similarly to the case of **ostringstream**, the example may look silly, since we may have done the last line directly with **cin**. The use of **istringstream** would be more reasonable when the given string does not come from the console or a file directly, but perhaps as an argument in a function, or from a database or graphical component, through a framework that passes the string as a whole.

However, unlike with the case of **ostringstream**, it can still be useful to resort to the seemingly silly technique above; keep in mind that if the given input does not exactly match what we request, **cin** reports an error, but it can also *jam* and could make the code to handle it a bit more involved. A commonly used trick when dealing directly with console input is to read with **getline** into a string variable (such that it reads *anything*), and then use an **istringstream** object to parse the line into the required pieces; if we get an error, we know that we can continue to read through **cin** without any issues.

Case Study: URL Encoding and Decoding

In this section, I will present a concrete application of the string manipulation facilities. The application uses both the C++ facilities, and the Boost Library regular expressions facilities.

I will present and discuss two programs, one that creates a URL-encoded version of a set of named parameters, and one that receives a URL-encoded string and decodes it (i.e., breaks it into individual pieces). In both cases, I use a simplified version of the URL-encoding mechanism, to keep the example simple.

URL-encoding is the mechanism used by web browsers to send form data to web servers for processing. The data consists of a set of values with given names. A simple example is an HTML login form, where we have two fields that could be named, for example, **`username`** and **`password`**. If, for instance, my username is **`carlos`** and my password is **`moreno`**, the browser would send the form data to the web server as the following URL-encoded string:

```
username=carlos&password=moreno
```

Spaces should be avoided (I will not discuss the reasons here, which are related to data communications principles), which means that if the values that the user types in contain spaces, these should be replaced by a non-whitespace character. In the case of URL-encoding, the character is the **`+`** sign. In the above example, if my password was **`moreno carlos`**, the URL-encoded string would be:

```
username=carlos&password=moreno+carlos
```

The simplified version of the URL-encoding mechanism that I will use for this case-study assumes that all input given by the user contains only alphanumeric characters in the US-English alphabet or spaces. This avoids the issue of special characters (for instance, what happens if the data entered by the user contains the character **`+`**? Or the character **`&`**, or **`=`**?). These issues are handled by the real URL-encoding technique, but I will avoid them in this case-study, to keep the example as simple as possible.

URL-Encoding Data Entered by the User

The first program prompts the user for pairs of data, indicating name of the field and value. The process stops when the user enters an empty name. At that point, the program outputs the URL-encoded data.

The basic structure of the program is a **`do-while`** loop, with stop condition given by an empty string read from the user. At each pass of the loop, the program reads the two pieces from the user and appends the corresponding item, in the form **`name=value`**, with proper encoding and separation with the character **`&`**. The program's structure for the URL-encoding is shown below:

```
string param_name, param_value, url_encoded;

do
{
    getline (cin, param_name);
    if (param_name != "")
    {
        getline (cin, param_value);
        //  ...
    }
}
while (param_name != "");

cout << "URL-encoded: " << url_encoded << endl;
```

Each individual encoding is done by iterating over the characters of the string and replacing it by its encoded version if necessary (replace spaces with **+**), as shown in the fragment below. We could also use the method **`find`** in a loop (while we keep finding spaces) and use the method **`replace`** at the positions where the space was found. However, this would be much more complicated than a simple for loop visiting each character of the string (and not much more efficient — surely, **`find`** will have to iterate over each character whilte searching for spaces).

```
for (string::size_type i = 0; i < text.length(); i++)
{
    if (text[i] == ' ')
    {
        text[i] = '+';
    }
}
```

The fragment of code below combines the two previous fragments (pseudocode and token encoding), completing the URL-encoding part of this case-study. It is assumed that the necessary variable declarations have been done (they are omitted below to save space). After the loop has completed, the variable **`url_encoded`** contains the result.

```
do
{
    getline (cin, param_name);
    if (param_name != "")
    {
        getline (cin, param_value);
        for (string::size_type i = 0; i < param_name.length(); i++)
        {
            if (param_name[i] == ' ')
            {
                param_name[i] = '+';
            }
        }
        for (string::size_type i = 0; i < param_value.length(); i++)
        {
            if (param_value[i] == ' ')
            {
                param_value = '+';
            }
        }
        if (url_encoded != "")    // Why this if?
        {
            url_encoded += '&';
        }
        url_encoded += param_name;
        url_encoded += '=';
        url_encoded += param_value;
    }
}
while (param_name != "");
```

Firefox Downloac Button

Brainteaser One: What happens if we omit the line **`if (url_encoded != "")`** and concatenate **`'&'`** directly? (this is an extension of the question in the comment for that line)

Brainteaser Two: Why two for loops? Can you re-arrange the code such that there is only one for loop? (and still works correctly, naturally!)

URL-Decoding Previously-encoded Data

The second program receives a string containing URL-encoded data and breaks it into pieces, displaying the pairs (parameter name and parameter value).

The basic idea is that we loop through the string while we keep finding occurences of the character **&**, which means that we keep finding additional parameter pairs. We have to keep track of consecutive occurences of this character, as this will allow us to extract the appropriate portion of the string, containing the parameter pair in the form `name=value`. The following fragment shows the basic structure of the program:

```
string encoded;

cout << "Enter a URL-encoded string: " << flush;
getline (cin, encoded);

string::size_type pos_start = 0, pos_end;
do
{
    pos_end = encoded.find ('&', pos_start);
    string param;
    if (pos_end != string::npos)
    {
        param = encoded.substr (pos_start, pos_end - pos_start);
        pos_start = pos_end + 1;
    }
    else
    {
        param = encoded.substr (pos_start);
    }

        // Break param into individual pieces
}
while (pos_end != string::npos);
```

Brainteaser: there is a bug in the code above, related to the last parameter pair in the string (and I bet at this point you know what I'm going to ask you to do).

We then break each parameter pair into individual components, `name` and `value`. To do this, we simply find the position of the equal sign, and then use it to determine the appropriate arguments for the two `substr`s, as shown below:

```
const string::size_type pos_eq = param.find ('=');
string name, value;
if (pos_eq  != string::npos)
{
    name = param.substr (0, pos_eq);
    value = param.substr (pos_eq + 1);
}
else
{
    // Error -- invalid parameter pair
}
```

We also have to replace the **+** characters back to spaces. We do this the same way as we did for token encoding, except that in this case we test if it is a **+** and replace it with a space.

Putting together all the pieces, we obtain the following program:

```
string encoded;

cout << "Enter a URL-encoded string: " << flush;
getline (cin, encoded);

string::size_type pos_start = 0, pos_end;
do
{
    pos_end = encoded.find ('&', pos_start);
    string param;
    if (pos_end != string::npos)
    {
        param = encoded.substr (pos_start, pos_end - pos_start);
        pos_start = pos_end + 1;
    }
    else
    {
        param = encoded.substr (pos_start);
    }

    for (string::size_type i = 0; i < param.length(); i++)
    {
        if (param[i] == '+')
        {
            param[i]  = ' ';
        }
    }

    const string::size_type pos_eq = param.find ('=');
    if (pos_eq  != string::npos)
    {
        const string name = param.substr (0, pos_eq);
        const string value = param.substr (pos_eq + 1);
        cout << name << " = " << value << endl;
    }
    else
    {
        cerr << "Invalid parameter found -- ignoring" << endl;
    }
}
while (pos_end != string::npos && pos_end != encoded.length() - 1);
```

Firefox Download Button