
13.1. Purpose of the String Classes

The string classes of the C++ standard library enable you to use strings as normal types that cause no problems for the user. Thus, you can copy, assign, and compare strings as fundamental types without worrying about whether there is enough memory or how long the internal memory is valid. You simply use operators, such as assignment by using `=`, comparison by using `==`, and concatenation by using `+`. In short, the string types of the C++ standard library are designed to behave as if they were a kind of fundamental data type that does not cause any trouble (at least in principle). Modern data processing is mostly string processing, so this is an important step for programmers coming from C, Fortran, or similar languages in which strings are a source of trouble.

The following sections offer two examples that demonstrate the abilities and uses of the string classes.

13.1.1. A First Example: Extracting a Temporary Filename

The first example program uses command-line arguments to generate temporary filenames. For example, if you start the program as

```
string1 prog.dat mydir hello. oops.tmp end.dat
```

the output is

```
prog.dat => prog.tmp
mydir => mydir.tmp
hello. => hello.tmp
oops.tmp => oops.xxx
end.dat => end.tmp
```

Usually, the generated filename has the extension `.tmp`, whereas the temporary filename for a name with the extension `.tmp` is `.xxx`.

The program is written in the following way:

Click here to view code image

```
// string/string1.cpp

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

int main (int argc, char* argv[])
{
    string filename, basename, extname, tmpname;
    const string suffix("tmp");

    // for each command-line argument (which is an ordinary C-string)
    for (int i=1; i<argc; ++i) {
        // process argument as filename
        filename = argv[i];

        // search period in filename
        string::size_type idx = filename.find('.');
        if (idx == string::npos) {
            // filename does not contain any period
            tmpname = filename + '.' + suffix;
        }
        else {
            // split filename into base name and extension
            // - base name contains all characters before the period
            // - extension contains all characters after the period
            basename = filename.substr(0, idx);
            extname = filename.substr(idx+1);
            if (extname.empty()) {
                // contains period but no extension: append tmp
                tmpname = filename;
                tmpname += suffix;
            }
            else if (extname == suffix) {
                // replace extension tmp with xxx
                tmpname = filename;
                tmpname.replace (idx+1, extname.size(), "xxx");
            }
```

```
            else {
                // replace any extension with tmp
                tmpname = filename;
                tmpname.replace (idx+1, string::npos, suffix);
            }
        }

        // print filename and temporary name
        cout << filename << " => " << tmpname << endl;
    }
}
```

At first, the header file for the C++ standard string classes is included:

```
#include <string>
```

As usual, these classes are declared in namespace `std` .

The following declaration creates four string variables:

```
string filename, basename, extname, tmpname;
```

No argument is passed, so the default constructor for `string` is called for their initialization. The default constructor initializes them as empty strings.

The following declaration creates a constant string `suffix` that is used in the program as the normal suffix for temporary filenames:

```
const string suffix("tmp");
```

The string is initialized by an ordinary C-string, so it has the value `tmp` . Note that C-strings can be combined with objects of class `string` in almost any situation in which two `string` s can be combined. In particular, in the entire program, every occurrence of `suffix` could be replaced with `"tmp"` so that a C-string is used directly.

In each iteration of the `for` loop, the following statement assigns a new value to the string variable `filename` :

```
filename = argv[i];
```

In this case, the new value is an ordinary C-string. However, it could also be another object of class `string` or a single character that has type `char` .

The following statement searches for the first occurrence of a period inside the string `filename` :

```
string::size_type idx = filename.find('.');
```

The `find()` function is one of several functions that search for something inside strings. You could also search backward, for substrings, only in a part of a string, or for more than one character simultaneously. All these find functions return an index of the first matching position. Yes, the return value is an integer and not an iterator. The usual interface for strings is not based on the concept of the STL. However, some iterator support for strings is provided (see Section 13.2.14, page 684). The return type of all find functions is `string::size_type` , an unsigned integral type that is defined inside the string class.[1] As usual, the index of the first character is the value `0` . The index of the last character is the value "*numberOfCharacters* `-1` ."

[1] In particular, the `size_type` of a string depends on the memory model of the string class. See Section 13.3.13, page 715, for details.

If the search fails, a special value is needed to return the failure. That value is `npos` , which is also defined by the string class. Thus, the following line checks whether the search for the period failed:

```
if (idx == string::npos)
```

The type and value of `npos` are a big pitfall for the use of strings. Be very careful that you always use `string::size_type` , *not* `int` or `unsigned` , for the return type when you want to check the return value of a find function. Otherwise, the comparison with `string::npos` might not work. See Section 13.2.12, page 680, for details.

If the search for the period fails in this example, the filename has no extension. In this case, the temporary filename is the concatenation of the original filename, the period character, and the previously defined extension for temporary files:

```
tmpname = filename + '.' + suffix;
```

Thus, you can simply use operator `+` to concatenate two strings. It is also possible to concatenate strings with ordinary C-strings and single characters.

If the period is found, the `else` part is used. Here, the index of the period is used to split the filename into a base part and the

extension. This is done by the `substr()` member function:

```
basename = filename.substr(0, idx);
extname = filename.substr(idx+1);
```

The first parameter of the `substr()` function is the starting index. The optional second argument is the number of characters, not the end index. If the second argument is not used, all remaining characters of the string are returned as a substring.

At all places where an index and a length are used as arguments, strings behave according to the following two rules:

1. An argument specifying the **index** must have a valid value. That value must be less than the number of characters of the string (as usual, the index of the first character is `0`). In addition, the index of the position after the last character could be used to specify the end.

   In most cases, a use of an index greater than the number of characters throws `out_of_range`. However, all functions that search for a character or a position allow any index. If the index exceeds the number of characters, these functions simply return `string::npos` ("not found").

2. An argument specifying the **number of characters** could have any value. If the size is greater than the remaining number of characters, all remaining characters are used. In particular, `string::npos` always works as a synonym for "all remaining characters."

Thus, the following expression throws an exception if the period is not found:

```
filename.substr(filename.find('.'))
```

But the following expression does not throw an exception:

```
filename.substr(0, filename.find('.'))
```

If the period is not found, it results in the whole filename.

Even if the period is found, the extension that is returned by `substr()` might be empty because there are no more characters after the period. This is checked by

```
if (extname.empty())
```

If this condition yields `true`, the generated temporary filename becomes the ordinary filename that has the normal extension appended:

```
tmpname = filename;
tmpname += suffix;
```

Here, operator `+=` is used to append the extension.

The filename might already have the extension for temporary files. To check this, operator `==` is used to compare two strings:

```
if (extname == suffix)
```

If this comparison yields `true`, the normal extension for temporary files is replaced by the extension `xxx`:

```
tmpname = filename;
tmpname.replace (idx+1, extname.size(), "xxx");
```

Here, the number of characters of the string `extname` is returned by

```
extname.size()
```

Instead of `size()`, you could use `length()`, which does exactly the same thing. So, both `size()` and `length()` return the number of characters. In particular, `size()` has nothing to do with the memory that the string uses.[2]

[2] In this case, two member functions do the same with respect to the two different design approaches that are merged here: `length()` returns the length of the string, just as `strlen()` does for ordinary C-strings, whereas `size()` is the common member function for the number of elements according to the concept of the STL.

Next, after all special conditions are considered, normal processing takes place. The program replaces the whole extension by the ordinary extension for temporary filenames:

```
tmpname = filename;
tmpname.replace (idx+1, string::npos, suffix);
```

Here, `string::npos` is used as a synonym for "all remaining characters." Thus, all remaining characters after the period are replaced with `suffix`. This replacement would also work if the filename contained a period but no extension. It would simply replace "nothing" with `suffix`.

The statement that writes the original filename and the generated temporary filename shows that you can print the strings by using the usual output operators of streams (surprise, surprise):

```
cout << filename << " => " << tmpname << endl;
```

13.1.2. A Second Example: Extracting Words and Printing Them Backward

The second example extracts single words from standard input and prints the characters of each word in reverse order. The words are separated by the usual whitespaces (newline, space, and tab) and by commas, periods, or semicolons:

Click here to view code image

```
// string/string2.cpp

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

int main (int argc, char** argv)
{
   const string delims(" \t,.;");
   string line;
   // for every line read successfully
   while (getline(cin,line)) {
       string::size_type begIdx, endIdx;

       // search beginning of the first word
       begIdx = line.find_first_not_of(delims);

       // while beginning of a word found
       while (begIdx != string::npos) {
           // search end of the actual word
           endIdx = line.find_first_of (delims, begIdx);
           if (endIdx == string::npos) {
               // end of word is end of line
               endIdx = line.length();
           }

           // print characters in reverse order
           for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
               cout << line[i];
           }
           cout << ' ';

           // search beginning of the next word
           begIdx = line.find_first_not_of (delims, endIdx);
       }
       cout << endl;
   }
}
```

In this program, all characters used as word separators are defined in a special string constant:

```
const string delims(" \t,.;");
```

The newline character is also used as a delimiter. However, no special processing is necessary for it because the program reads line by line.

The outer loop runs as far as a line can be read into the string `line` :

```
string line;
while (getline(cin,line)) {
    ...
}
```

The function `getline()` is a special function to read input from streams into a string. It reads every character up to the next end-of-line, which by default is the newline character. The line delimiter itself is extracted but not appended. By passing your special line delimiter as an optional third character argument, you can use `getline()` to read token by token, where the tokens are separated by that special delimiter.

Inside the outer loop, the individual words are searched and printed. The first statement searches for the beginning of the first word:

```
begIdx = line.find_first_not_of(delims);
```

The `find_first_not_of()` function returns the first index of a character that is not part of the passed string argument. Thus, this function returns the position of the first character that is not one of the separators in `delims` . As usual for find functions, if no matching index is found, `string::npos` is returned.

The inner loop iterates as long as the beginning of a word can be found:

```
while (begIdx != string::npos) {
    ...
}
```

The first statement of the inner loop searches for the end of the current word:

```
endIdx = line.find_first_of (delims, begIdx);
```

The `find_first_of()` function searches for the first occurrence of one of the characters passed as the first argument. In this case, an optional second argument is used that specifies where to start the search in the string. Thus, the first delimiter after the beginning of the word is searched. If no such character is found, the end-of-line is used:

```
if (endIdx == string::npos) {
    endIdx = line.length();
}
```

Here, `length()` is used, which does the same thing as `size()` : It returns the number of characters.

In the next statement, all characters of the word are printed in reverse order:

```
for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
    cout << line[i];
}
```

Accessing a single character of the string is done with operator `[]` . Note that this operator does *not* check whether the index of the string is valid. Thus, the programmer has to ensure that the index is valid, as was done here. A safer way to access a character is to use the `at()` member function. However, such a check costs runtime, so the check is not provided for the usual accessing of characters of a string.

Note that for operator `[]` , the number of characters is a valid index, returning a character representing the end of the string. This *end-of-string* character is initialized by the default constructor of the character type (`'\0'` for class `string`):[3]

[3] Before C++11, for the nonconstant version of operator `[]` , the current number of characters was an invalid index. Using it did result in undefined behavior.

```
string s;
s[s.length()]      // yields '\0'
```

Another nasty problem results from using the index of the string. That is, if you omit the cast of `begIdx` to `int` , this program might run in an endless loop or might crash. Similar to the first example program, the problem is that `string::size_type` is an unsigned integral type. Without the cast, the signed value `i` is converted automatically into an unsigned value because it is compared with a unsigned type. In this case, the following expression always yields `true` if the current word starts at the beginning of the line:

```
i>=begIdx
```

The reason is that `begIdx` is then `0` , and any unsigned value is greater than or equal to `0` . So, an endless loop results that might get stopped by a crash due to an illegal memory access. For this reason, I don't like the concept of `string::size_type` and `string::npos` . See Section 13.2.12, page 681, for a workaround that is safer, but not perfect.

The last statement of the inner loop reinitializes `begIdx` to the beginning of the next word, if any:

```
begIdx = line.find_first_not_of (delims, endIdx);
```

Here, unlike with the first call of `find_first_not_of()` in the example, the end of the previous word is passed as the starting index for the search. If the previous word was the rest of the line, `endIdx` is the index of the end of the line. This simply means that the search starts from the end of the string, which returns `string::npos` .

Let's try this "useful and important" program. Here is some possible input:[4]

[4] Thanks to Sean Okeefe for providing the last two lines.

```
pots & pans
I saw a reed
deliver no pets
nametag on diaper
```

The output for this input is as follows:

```
stop & snap
I was a deer
```

```
reviled on step
gateman no repaid
```