# String_Ref

Marshall Clow



## Table of Contents

# Overview

Boost.StringRef is an implementation of Jeffrey Yaskin's N3442: string_ref: a non-owning reference to a string.

When you are parsing/processing strings from some external source, frequently you want to pass a piece of text to a procedure for specialized processing. The canonical way to do this is as a `std::string`, but that has certain drawbacks:

1) If you are processing a buffer of text (say a HTTP response or the contents of a file), then you have to create the string from the text you want to pass, which involves memory allocation and copying of data.

2) if a routine receives a constant `std::string` and wants to pass a portion of that string to another routine, then it must create a new string of that substring.

3) A routine receives a constant `std::string` and wants to return a portion of the string, then it must create a new string to return.

`string_ref` is designed to solve these efficiency problems. A `string_ref` is a read-only reference to a contiguous sequence of characters, and provides much of the functionality of `std::string`. A `string_ref` is cheap to create, copy and pass by value, because it does not actually own the storage that it points to.

A `string_ref` is implemented as a small struct that contains a pointer to the start of the character data and a count. A `string_ref` is cheap to create and cheap to copy.

`string_ref` acts as a container; it includes all the methods that you would expect in a container, including iteration support, `operator []`, `at` and `size`. It can be used with any of the iterator-based algorithms in the STL - as long as you don't need to change the underlying data (`sort` and `remove`, for example, will not work)

Besides generic container functionality, `string_ref` provides a subset of the interface of `std::string`. This makes it easy to replace parameters of type `const std::string &` with `boost::string_ref`. Like `std::string`, `string_ref` has a static member variable named `npos` to denote the result of failed searches, and to mean "the end".

Because a `string_ref` does not own the data that it "points to", it introduces lifetime issues into code that uses it. The programmer must ensure that the data that a `string_ref` refers to exists as long as the `string_ref` does.

# Examples

Integrating `string_ref` into your code is fairly simple. Wherever you pass a `const std::string &` or `std::string` as a parameter, that's a candidate for passing a `boost::string_ref`.

```
std::string extract_part ( const std::string &bar ) {
    return bar.substr ( 2, 3 );
    }

if ( extract_part ( "ABCDEFG" ).front() == 'C' ) { /* do something */ }
```

Let's figure out what happens in this (contrived) example.

First, a temporary string is created from the string literal `"ABCDEFG"`, and it is passed (by reference) to the routine `extract_part`. Then a second string is created in the call `std::string::substr` and returned to `extract_part` (this copy may be elided by RVO). Then `extract_part` returns that string back to the caller (again this copy may be elided). The first temporary string is deallocated, and `front` is called on the second string, and then it is deallocated as well.

Two `std::string`s are created, and two copy operations. That's (potentially) four memory allocations and deallocations, and the associated copying of data.

Now let's look at the same code with `string_ref`:

```
boost::string_ref extract_part ( boost::string_ref bar ) {
    return bar.substr ( 2, 3 );
    }

if ( extract_part ( "ABCDEFG" ).front() == "C" ) { /* do something */ }
```

No memory allocations. No copying of character data. No changes to the code other than the types. There are two `string_ref`s created, and two `string_ref`s copied, but those are cheap operations.

# Reference

The header file "string_ref.hpp" defines a template `boost::basic_string_ref`, and four specializations - for `char` / `wchar_t` / `char16_t` / `char32_t` .

```
#include <boost/utility/string_ref.hpp>
```

Construction and copying:

```
BOOST_CONSTEXPR basic_string_ref ();    // Constructs an empty string_ref
BOOST_CONSTEXPR basic_string_ref(const charT* str); // Constructs from a NULL-terminated string
BOOST_CONSTEXPR basic_string_ref(const charT* str, size_type len); // Constructs from a point↵
er, length pair
template<typename Allocator>
basic_string_ref(const std::basic_string<charT, traits, Allocator>& str); // Constructs from a ↵
std::string
basic_string_ref (const basic_string_ref &rhs);
basic_string_ref& operator=(const basic_string_ref &rhs);
```

`string_ref` does not define a move constructor nor a move-assignment operator because copying a `string_ref` is just a cheap as moving one.

Basic container-like functions:

```
BOOST_CONSTEXPR size_type size()     const ;
BOOST_CONSTEXPR size_type length()   const ;
BOOST_CONSTEXPR size_type max_size() const ;
BOOST_CONSTEXPR bool empty()         const ;

// All iterators are const_iterators
BOOST_CONSTEXPR const_iterator  begin() const ;
BOOST_CONSTEXPR const_iterator cbegin() const ;
BOOST_CONSTEXPR const_iterator    end() const ;
BOOST_CONSTEXPR const_iterator   cend() const ;
const_reverse_iterator         rbegin() const ;
const_reverse_iterator        crbegin() const ;
const_reverse_iterator           rend() const ;
const_reverse_iterator          crend() const ;
```

Access to the individual elements (all of which are const):

```
BOOST_CONSTEXPR const charT& operator[](size_type pos) const ;
const charT& at(size_t pos) const ;
BOOST_CONSTEXPR const charT& front() const ;
BOOST_CONSTEXPR const charT& back()  const ;
BOOST_CONSTEXPR const charT* data()  const ;
```

Modifying the `string_ref` (but not the underlying data):

```
void clear();
void remove_prefix(size_type n);
void remove_suffix(size_type n);
```

Searching:

```
size_type find(basic_string_ref s) const ;
size_type find(charT c) const ;
size_type rfind(basic_string_ref s) const ;
size_type rfind(charT c) const ;
size_type find_first_of(charT c) const ;
size_type find_last_of (charT c) const ;

size_type find_first_of(basic_string_ref s) const ;
size_type find_last_of(basic_string_ref s) const ;
size_type find_first_not_of(basic_string_ref s) const ;
size_type find_first_not_of(charT c) const ;
size_type find_last_not_of(basic_string_ref s) const ;
size_type find_last_not_of(charT c) const ;
```

String-like operations:

```
BOOST_CONSTEXPR basic_string_ref substr(size_type pos, size_type n=npos) const ; // Creates a ↵
new string_ref
bool starts_with(charT c) const ;
bool starts_with(basic_string_ref x) const ;
bool ends_with(charT c) const ;
bool ends_with(basic_string_ref x) const ;
```

# History

## boost 1.53

- Introduced