

Document number: PxxxxR0  
Date: 2019-05-23  
Project: LWG  
Reply-to: Paul Fee <paul.f.fee@gmail.com>

# String substring checking

## 1 Abstract

This paper proposes to add a `contains` member function to the class templates `basic_string` and `basic_string_view`. This function will check whether or not a string contains a substring.

## 2 History

### 2.1 R0

- Initial version

## 3 Motivation

Checking whether or not a string contains a substring is a common task. Standard libraries of many other programming languages provide routines for performing this check, for example:

- Python: operator `in` which calls an object's `__contains__(self, item)` method <sup>1</sup>.
- Java: class `String` has a `contains` method <sup>2</sup>.
- Rust: `struct std::str` and `struct std::string::String` have `contains` methods. <sup>3</sup>

Also, some C++ libraries (other than the standard library) that implement a string type include such methods. For example, Qt library has classes `QString` and `QStringRef` (analogous to `std::string_view`) which have `contains` member functions <sup>4 5</sup>.

The source code of LLVM includes a `StringRef` class with a `contains` method similar to that proposed here.

These functions are widely used. For example, the source code for Qt 5.12.3 has 8364 occurrences of `contains`, although these include methods of classes such as `QList`.

---

1 Python Language Reference, Expressions, Membership test operations:  
<https://docs.python.org/3/reference/expressions.html#in>

2 Java 2 Platform SE 5.0, Class `String`:  
[https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#contains\(java.lang.CharSequence\)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#contains(java.lang.CharSequence))

3 The Rust Standard Library: <https://doc.rust-lang.org/std/primitive.str.html#method.contains> <https://doc.rust-lang.org/std/string/struct.String.html#method.contains>

4 Qt Core, `QString::contains()` <https://doc.qt.io/qt-5/qstring.html#contains>

5 Qt Core, `QStringRef::contains()` <https://doc.qt.io/qt-5/qstringref.html#contains>

## 4 Prior work

The `basic_string` and `basic_string_view` class templates gained `starts_with` and `ends_with` methods in C++2a. A `contains` method would complement those two methods<sup>6</sup>.

### 4.1 Existing substring checks vs proposal

A range of options exist for substring checking.

Using the C library:

```
std::string haystack = "no place for needles";  
if (strstr(haystack.c_str(), "needle"))
```

Using the C++ standard library:

```
if (haystack.find("needle") != std::string::npos)
```

Using Boost algorithms library<sup>7</sup>:

```
if (boost::contains(haystack, "needle"))
```

The proposed changes would provide a concise, unambiguous method for substring checking in which the intent is clearly expressed.

```
if (haystack.contains("needle"))
```

## 5 Design considerations

### 5.1 Standard library vs core language change

Python uses the `in` operator, such as:

```
if 'needle' in haystack:
```

Adopting a similar approach in C++ would involve a new keyword. A new keyword risks breaking backwards compatibility with code already using ‘`in`’ for other purposes, such as variable names. Hence changes to the standard library are preferred.

### 5.2 Member function vs free function

This proposal adds the `contains` member function to the `basic_string` and `basic_string_view` class templates. Another option considered was to add a free function to the namespace `std`. However adding a member function is consistent with the existing API for `starts_with` and `ends_with`.

In addition to API consistency, a free function is ambiguous (`contains(string, substring)` vs `contains(substring, string)`).

A member function also offers consistency with string classes in other popular languages and C++ projects [2 3 4 5].

---

<sup>6</sup> WG21, p0457: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0457r0.html>

<sup>7</sup> Boost algorithm library: <https://www.boost.org/doc/libs/release/doc/html/boost/algorithm/contains.html>

## 5.3 Member function name

### 5.3.1 Matching name used by containers

In C++2a, the `set`, `multiset`, `map` and `multimap` containers gained a `contains` method. For the `multiset` and `multimap` containers, this offers a performance boost over `count` as multiple matches need not be counted. However for `map` and `set`, the benefit is primarily clearer expression of intent in the calling code, with `contains` returning a `bool` compared to `count` returning a `size_type` of value 0 or 1.

With the `basic_string` and `basic_string_view` class templates, the proposed `contains` method is not directly analogous to the `contains` method applied to a container. Rather than to search for a key, a `contains` operation on a string means to search for a substring.

Since the proposal adds a `contains` member function to the `basic_string` and `basic_string_view` class templates, the context of the operation is implicit. This reuse of function name is seen elsewhere, taking Qt as an example, both the `QString` and `QList`<sup>8</sup> classes have `contains` member functions.

Similarly, Python uses the `in` operator for both substring checking and container searches. The intent is clear in each context, hence `contains` for two types of operation should not cause confusion for C++ users.

### 5.3.2 Alternatives to contains

Naming the member function `within` would avoid matching a function name used by containers.

```
std::string haystack = "no place for needles";
std::string substring = "needle";
if (substring.within(haystack))
```

A `within` member function would not be directly usable with string literals, instead needing a `string` or `string_view` object.

```
if ("needle".within(haystack))    // Compilation error
if ("needle"s.within(haystack))   // using namespace std::literals
if ("needle"sv.within(haystack)) // using namespace std::literals
```

However a member function named `contains` matches the approach used in other languages and C++ projects. It also accepts string literals without needing `std::literals`.

```
std::string haystack = "no place for needles";
if (haystack.contains("needle"))
```

## 5.4 Case insensitivity

The `starts_with` and `ends_with` member functions do not have any case awareness. Likewise the proposed `contains` member function provides a case sensitive substring check.

The `starts_with`, `ends_with` and `contains` member functions could be extended via future proposals to include case insensitive operations.

---

<sup>8</sup> Qt Core, `QList::contains()` <https://doc.qt.io/qt-5/qlist.html#contains>

Some C++ libraries provide a case insensitive substring check. For example, Boost strings algorithms provide `icontains`<sup>9</sup>. Qt's `QString::contains` method takes a parameter that defaults to case sensitive, but allows case insensitivity to be specified.

For example, with Boost:

```
std::string haystack = "no place for needles";
if (icontains(haystack, "Needle"))
```

and with Qt:

```
QString haystack = "no place for needles";
if (haystack.contains("Needle", Qt::CaseInsensitive));
```

If the proposed `contains` member function were to support case insensitivity, then an additional parameter, that defaulted to case sensitive would be preferred. A possible member function declaration could be:

```
enum case_sensitivity {
    case_sensitive;
    case_insensitive;
};

bool contains(basic_string_view<charT, traits> s,
              case_sensitivity cs = case_sensitive,
              const locale& loc = std::locale()) const noexcept;
```

However case sensitivity is a complex topic for character sets beyond ASCII. Therefore, while not ruling out case insensitivity entirely, the scope of this proposal is limited to case sensitive substring checks.

## 5.5 Function overloads

The proposal for `starts_with` and `ends_with` included three overloads per function, per class. This proposal has the same set of overloads.

```
// basic_string:
bool contains(charT c) const noexcept;
bool contains(basic_string_view<charT, traits> s) const noexcept;
bool contains(const charT* s) const noexcept;

// basic_string_view:
constexpr bool contains(charT c) const noexcept;
constexpr bool contains(basic_string_view<charT, traits> s) const noexcept;
constexpr bool contains(const charT* s) const noexcept;
```

Within both templates, an overload accepting a `basic_string` is not required since `basic_string` has a non-explicit conversion operator to `basic_string_view`.

---

9 Boost algorithm library: [https://www.boost.org/doc/libs/1\\_70\\_0/doc/html/boost/algorithm/icontains.html](https://www.boost.org/doc/libs/1_70_0/doc/html/boost/algorithm/icontains.html)

## 5.6 Passing by value vs passing by reference

In accordance with the guidance from P0254R1<sup>10</sup>, objects of type `basic_string_view` are passed by value (not by reference).

## 5.7 Possible implementation

Using `libstdc++` as an example, one possible implementation of `contains` could be:

```
constexpr bool  
basic_string_view::contains(basic_string_view __x) const noexcept  
{ return this->find(__x) != npos; }
```

The remaining overloads would follow the same approach of calling `find` and checking against `npos`. The complexity of the `contains` member function would be equivalent to that of the `find` member function.

## 6 Wording

[To be completed]

---

<sup>10</sup> Integrating `std::string_view` and `std::string`,  
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0254r1.pdf>