```
    REQUIRE(word.substr() == "hobbits"); ❷
  }
  SECTION("position takes the remainder") {
    REQUIRE(word.substr(3) == "bits"); ❸
  }
  SECTION("position/index takes a substring") {
    REQUIRE(word.substr(3, 3) == "bit"); ❹
  }
}
```

*Listing 15-12: Extracting substrings from a string*

You declare a string called word containing hobbits ❶. If you invoke substr with no arguments, you simply copy the string ❷. When you provide the position argument 3, substr extracts the substring beginning at element 3 and extending to the end of the string, yielding bits ❸. Finally, when you provide a position (3) and a length (3), you instead get bit ❹.

### Summary of string Manipulation Methods

Table 15-5 lists many of the insertion and deletion methods of string. In this table, str is a string or a C-style char* string, p and n are size_t, ind is a size_t index or an iterator into s, n and i are a size_t, c is a char, and beg and end are iterators. An asterisk (*) indicates that this operation invalidates raw pointers and iterators to v's elements in at least some circumstances.

**Table 15-5:** Supported std::string Element Manipulation Methods

| Method | Description |
|---|---|
| s.insert(ind, str, [p], [n]) | Inserts the n elements of str, starting at p, into s just before ind. If no n supplied, inserts the entire string or up to the first null of a char*; p defaults to 0.* |
| s.insert(ind, n, c) | Inserts n copies of c just before ind.* |
| s.insert(ind, beg, end) | Inserts the half-open range from beg to end just before ind. * |
| s.append(str, [p], [n]) | Equivalent to s.insert(s.end(), str, [p], [n]).* |
| s.append(n, c) | Equivalent to s.insert(s.end(), n, c).* |
| s.append(beg, end) | Appends the half-open range from beg to end to the end of s.* |
| s += c<br>s += str | Appends c or str to the end of s.* |
| s.push_back(c) | Appends c to the end of s.* |
| s.clear() | Removes all characters from s.* |
| s.erase([i], [n]) | Removes n characters starting at position i; i defaults to 0, and n defaults to the remainder of s.* |
| s.erase(itr) | Erases the element pointed to by itr.* |
| s.erase(beg, end) | Erases the elements on the half-open range from beg to end.* |
| s.pop_back() | Removes the last element of s.* |

*(continued)*

**Table 15-5:** Supported `std::string` Element Manipulation Methods (continued)

| Method | Description |
|---|---|
| `s.resize(n,[c])` | Resizes the string so it contains **n** characters. If this operation increases the string's length, it adds copies of **c**, which defaults to 0.* |
| `s.replace(i, n1, str, [p], [n2])` | Replaces the **n1** characters starting at index **i** with the **n2** elements in **str** starting at **p**. By default, **p** is 0 and **n2** is `str.length()`.* |
| `s.replace(beg, end, str)` | Replaces the half-open range **beg** to **end** with **str**.* |
| `s.replace(p, n, str)` | Replaces from index **p** to **p+n** with **str**.* |
| `s.replace(beg1, end1, beg2, end2)` | Replaces the half-open range **beg1** to **end1** with the half-open range **beg2** to **end2**.* |
| `s.replace(ind, c, [n])` | Replaces **n** elements starting at **ind** with **c**s.* |
| `s.replace(ind, beg, end)` | Replaces elements starting at **ind** with the half-open range **beg** to **end**.* |
| `s.substr([p], [c])` | Returns the substring starting at **p** with length **c**. By default, **p** is 0 and **c** is the remainder of the string. |
| `s1.swap(s2)`<br>`swap(s1, s2)` | Exchanges the contents of **s1** and **s2**.* |

## Search

In addition to the preceding methods, `string` offers several *search methods*, which enable you to locate substrings and characters that you're interested in. Each method performs a particular kind of search, so which you choose depends on the particulars of the application.

### find

The first method `string` offers is `find`, which accepts a `string`, a C-style `string`, or a `char` as its first argument. This argument is an element that you want to locate within `this`. Optionally, you can provide a second `size_t` position argument that tells `find` where to start looking. If `find` fails to locate the substring, it returns the special `size_t`-valued, constant, `static` member `std::string::npos`. Listing 15-13 illustrates the `find` method.

```
TEST_CASE("std::string find") {
  using namespace std::literals::string_literals;
  std::string word("pizzazz"); ❶
  SECTION("locates substrings from strings") {
    REQUIRE(word.find("zz"s) == 2); // pi(z)zazz ❷
  }
  SECTION("accepts a position argument") {
    REQUIRE(word.find("zz"s, 3) == 5); // pizza(z)z ❸
  }
  SECTION("locates substrings from char*") {
    REQUIRE(word.find("zaz") == 3); // piz(z)azz ❹
  }
```

```
  SECTION("returns npos when not found") {
    REQUIRE(word.find('x') == std::string::npos); ❺
  }
}
```

*Listing 15-13: Finding substrings within a* `string`

Here, you construct the `string` called `word` containing `pizzazz` ❶. In the first test, you invoke `find` with a `string` containing `zz`, which returns `2` ❷, the index of the first *z* in `pizzazz`. When you provide a position argument of `3` corresponding to the second *z* in `pizzazz`, `find` locates the second *zz* beginning at `5` ❸. In the third test, you use the C-style string `zaz`, and `find` returns 3, again corresponding to the second *z* in `pizzazz` ❹. Finally, you attempt to find the character *x*, which doesn't appear in `pizzazz`, so `find` returns `std::string::npos` ❺.

## rfind

The `rfind` method is an alternative to `find` that takes the same arguments but searches *in reverse*. You might want to use this functionality if, for example, you were looking for particular punctuation at the end of a `string`, as Listing 15-14 illustrates.

```
TEST_CASE("std::string rfind") {
  using namespace std::literals::string_literals;
  std::string word("pizzazz"); ❶
  SECTION("locates substrings from strings") {
    REQUIRE(word.rfind("zz"s) == 5); // pizza(z)z ❷
  }
  SECTION("accepts a position argument") {
    REQUIRE(word.rfind("zz"s, 3) == 2); // pi(z)zazz ❸
  }
  SECTION("locates substrings from char*") {
    REQUIRE(word.rfind("zaz") == 3); // piz(z)azz ❹
  }
  SECTION("returns npos when not found") {
    REQUIRE(word.rfind('x') == std::string::npos); ❺
  }
}
```

*Listing 15-14: Finding substrings in reverse within a* `string`

Using the same `word` ❶, you use the same arguments as in Listing 15-13 to test `rfind`. Given `zz`, `rfind` returns `5`, the second to last *z* in `pizzazz` ❷. When you provide the positional argument `3`, `rfind` instead returns the first *z* in `pizzazz` ❸. Because there's only one occurrence of the substring `zaz`, `rfind` returns the same position as `find` ❹. Also like `find`, `rfind` returns `std::string::npos` when given `x` ❺.

## find_*_of

Whereas `find` and `rfind` locate exact subsequences in a `string`, a family of related functions finds the first character contained in a given argument.

The `find_first_of` function accepts a `string` and locates the first character in this contained in the argument. Optionally, you can provide a `size_t` position argument to indicate to `find_first_of` where to start in the string. If `find_first_of` cannot find a matching character, it will return `std::string::npos`. Listing 15-15 illustrates the `find_first_of` function.

```
TEST_CASE("std::string find_first_of") {
  using namespace std::literals::string_literals;
  std::string sentence("I am a Zizzer-Zazzer-Zuzz as you can plainly see."); ❶
  SECTION("locates characters within another string") {
    REQUIRE(sentence.find_first_of("Zz"s) == 7); // (Z)izzer ❷
  }
  SECTION("accepts a position argument") {
    REQUIRE(sentence.find_first_of("Zz"s, 11) == 14); // (Z)azzer ❸
  }
  SECTION("returns npos when not found") {
    REQUIRE(sentence.find_first_of("Xx"s) == std::string::npos); ❹
  }
}
```

*Listing 15-15: Finding the first element from a set within a `string`*

The `string` called `sentence` contains `I am a Zizzer-Zazzer-Zuzz as you can plainly see.` ❶. Here, you invoke `find_first_of` with the string `Zz`, which matches both lowercase and uppercase *z*. This returns `7`, which corresponds to the first `Z` in `sentence`, **`Z`**`izzer` ❷. In the second test, you again provide the string `Zz` but also pass the position argument `11`, which corresponds to the `e` in `Zizz`**`e`**`r`. This results in `14`, which corresponds to the `Z` in **`Z`**`azzer` ❸. Finally, you invoke `find_first_of` with `Xx`, which results in `std::string::npos` because `sentence` doesn't contain an `x` (or an `X`) ❹.

A `string` offers three `find_first_of` variations:

- `find_first_not_of` returns the first character *not* contained in the `string` argument. Rather than providing a `string` containing the elements you want to find, you provide a `string` of characters you *don't* want to find.
- `find_last_of` performs matching in reverse; rather than searching from the beginning of the `string` or from the position argument and proceeding to the end, `find_last_of` begins at the end of the `string` or from the position argument and proceeds to the beginning.
- `find_last_not_of` combines the two prior variations: you pass a `string` containing elements you don't want to find, and `find_last_not_of` searches in reverse.

Your choice of `find` function boils down to what your algorithmic requirements are. Do you need to search from the back of a `string`, say for a punctuation mark? If so, use `find_last_of`. Are you looking for the first space in a `string`? If so, use `find_first_of`. Do you want to invert your search and look for the first element that is not a member of some set? Then use the alternatives `find_first_not_of` and `find_last_not_of`, depending on whether you want to start from the beginning or end of the `string`.

Listing 15-16 illustrates these three `find_first_of` variations.

```
TEST_CASE("std::string") {
  using namespace std::literals::string_literals;
  std::string sentence("I am a Zizzer-Zazzer-Zuzz as you can plainly see."); ❶
  SECTION("find_last_of finds last element within another string") {
    REQUIRE(sentence.find_last_of("Zz"s) == 24); // Zuz(z) ❷
  }
  SECTION("find_first_not_of finds first element not within another string") {
    REQUIRE(sentence.find_first_not_of(" -IZaeimrz"s) == 22); // Z(u)zz ❸
  }
  SECTION("find_last_not_of finds last element not within another string") {
    REQUIRE(sentence.find_last_not_of(" .es"s) == 43); // plainl(y) ❹
    }
}
```

*Listing 15-16: Alternatives to the `find_first_of` method of `string`*

Here, you initialize the same `sentence` as in Listing 15-15 ❶. In the first test, you use `find_last_of` on `Zz`, which searches in reverse for any *z* or *Z* and returns `24`, the last *z* in the sentence `Zuzz` ❷. Next, you use `find_first_not_of` and pass a farrago of characters (not including the letter *u*), which results in `22`, the position of the first *u* in `Zuzz` ❸. Finally, you use `find_last_not_of` to find the last character not equal to space, period, *e*, or *s*. This results in `43`, the position of *y* in `plainly` ❹.

### Summary of string Search Methods

Table 15-6 lists many of the search methods for `string`. Note that `s2` is a string; `cstr` is a C-style `char*` string; `c` is a `char`; and `n`, `l`, and `pos` are `size_t` in the table.

**Table 15-6:** Supported `std::string` Search Algorithms

| Method | Searches `s` starting at `p` and returns the position of the . . . |
| --- | --- |
| `s.find(`**`s2`**`, [`**`p`**`])` | First substring equal to **`s2`**; **`p`** defaults to 0. |
| `s.find(`**`cstr`**`, [`**`p`**`], [`**`l`**`])` | First substring equal to the first **`l`** characters of **`cstr`**; **`p`** defaults to 0; **`l`** defaults to **`cstr`**'s length per null termination. |
| `s.find(`**`c`**`, [`**`p`**`])` | First character equal to **`c`**; **`p`** defaults to 0. |
| `s.rfind(`**`s2`**`, [`**`p`**`])` | Last substring equal to **`s2`**; **`p`** defaults to `npos`. |
| `s.rfind(`**`cstr`**`, [`**`p`**`], [`**`l`**`])` | Last substring equal to the first **`l`** characters of **`cstr`**; **`p`** defaults to `npos`; **`l`** defaults to **`cstr`**'s length per null termination. |
| `s.rfind(`**`c`**`, [`**`p`**`])` | Last character equal to **`c`**; **`p`** defaults to `npos`. |
| `s.find_first_of(`**`s2`**`, [`**`p`**`])` | First character contained in **`s2`**; **`p`** defaults to 0. |
| `s.find_first_of(`**`cstr`**`, [`**`p`**`], [`**`l`**`])` | First character contained in the first **`l`** characters of **`cstr`**; **`p`** defaults to 0; **`l`** defaults to **`cstr`**'s length per null termination. |

*(continued)*

**Table 15-6:** Supported `std::string` Search Algorithms (continued)

| Method | Searches s starting at p and returns the position of the . . . |
|---|---|
| `s.find_first_of(c, [p])` | First character equal to **c**; **p** defaults to 0. |
| `s.find_last_of(s2, [p])` | Last character contained in **s2**; **p** defaults to 0. |
| `s.find_last_of(cstr, [p], [l])` | Last character contained in the first **l** characters of **cstr**; **p** defaults to 0; **l** defaults to **cstr**'s length per null termination. |
| `s.find_last_of(c, [p])` | Last character equal to **c**; **p** defaults to 0. |
| `s.find_first_not_of(s2, [p])` | First character not contained in **s2**; **p** defaults to 0. |
| `s.find_first_not_of(cstr, [p], [l])` | First character not contained in the first **l** characters of **cstr**; **p** defaults to 0; **l** defaults to **cstr**'s length per null termination. |
| `s.find_first_not_of(c, [p])` | First character not equal to **c**; **p** defaults to 0. |
| `s.find_last_not_of(s2, [p])` | Last character not contained in **s2**; **p** defaults to 0. |
| `s.find_last_not_of(cstr, [p], [l])` | Last character not contained in the first **l** characters of **cstr**; **p** defaults to 0; **l** defaults to **cstr**'s length per null termination. |
| `s.find_last_not_of(c, [p])` | Last character not equal to **c**; **p** defaults to 0. |

### Numeric Conversions

The STL provides functions for converting between `string` or `wstring` and the fundamental numeric types. Given a numeric type, you can use the `std::to_string` and `std::to_wstring` functions to generate its `string` or `wstring` representation. Both functions have overloads for all the numeric types. Listing 15-17 illustrates `string` and `wstring`.

```
TEST_CASE("STL string conversion function") {
  using namespace std::literals::string_literals;
  SECTION("to_string") {
    REQUIRE("8675309"s == std::to_string(8675309)); ❶
  }
  SECTION("to_wstring") {
    REQUIRE(L"109951.1627776"s == std::to_wstring(109951.1627776)); ❷
  }
}
```

*Listing 15-17: Numeric conversion functions of `string`*

**NOTE** *Thanks to the inherent inaccuracy of the double type, the second unit test ❷ might fail on your system.*

The first example uses `to_string` to convert the `int 8675309` into a `string` ❶; the second example uses `to_wstring` to convert the `double` `109951.1627776` into a `wstring` ❷.

You can also convert the other way, going from a string or wstring to a numeric type. Each numeric conversion function accepts a string or wstring containing a string-encoded number as its first argument. Next, you can provide an optional pointer to a size_t. If provided, the conversion function will write the index of the last character it was able to convert (or the length of the input string if it decoded all characters). By default, this index argument is nullptr, in which case the conversion function doesn't write the index. When the target type is integral, you can provide a third argument: an int corresponding to the base of the encoded string. This base argument is optional and defaults to 10.

Each conversion function throws std::invalid_argument if no conversion could be performed and throws std::out_of_range if the converted value is out of range for the corresponding type.

Table 15-7 lists each of these conversion functions along with its target type. In this table, s is a string. If p is not nullptr, the conversion function will write the position of the first unconverted character in s to the memory pointed to by p. If all characters are encoded, returns the length of s. Here, b is the number's base representation in s. Note that p defaults to nullptr, and b defaults to 10.

**Table 15-7:** Supported Numeric Conversion Functions for std::string and std::wstring

| Function | Converts s to |
|---|---|
| stoi(**s**, [**p**], [**b**]) | An int |
| stol(**s**, [**p**], [**b**]) | A long |
| stoll(**s**, [**p**], [**b**]) | A long long |
| stoul(**s**, [**p**], [**b**]) | An unsigned long |
| stoull(**s**, [**p**], [**b**]) | An unsigned long long |
| stof(**s**, [**p**]) | A float |
| stod(**s**, [**p**]) | A double |
| stold(**s**, [**p**]) | A long double |
| to_string(**n**) | A string |
| to_wstring(**n**) | A wstring |

Listing 15-18 illustrates several numeric conversion functions.

```
TEST_CASE("STL string conversion function") {
  using namespace std::literals::string_literals;
  SECTION("stoi") {
    REQUIRE(std::stoi("8675309"s) == 8675309); ❶
  }
  SECTION("stoi") {
    REQUIRE_THROWS_AS(std::stoi("1099511627776"s), std::out_of_range); ❷
  }
  SECTION("stoul with all valid characters") {
    size_t last_character{};
    const auto result = std::stoul("0xD3C34C3D"s, &last_character, 16); ❸
```

```
    REQUIRE(result == 0xD3C34C3D);
    REQUIRE(last_character == 10);
  }
  SECTION("stoul") {
    size_t last_character{};
    const auto result = std::stoul("42six"s, &last_character); ❹
    REQUIRE(result == 42);
    REQUIRE(last_character == 2);
  }
  SECTION("stod") {
    REQUIRE(std::stod("2.7182818"s) == Approx(2.7182818)); ❺
  }
}
```

*Listing 15-18: String conversion functions of* `string`

First, you use `stoi` to convert `8675309` to an integer ❶. In the second test, you attempt to use `stoi` to convert the `string` `1099511627776` into an integer. Because this value is too large for an `int`, `stoi` throws `std::out_of_range` ❷. Next, you convert `0xD3C34C3D` with `stoi`, but you provide the two optional arguments: a pointer to a `size_t` called `last_character` and a hexadecimal base ❸. The `last_character` object is `10`, the length of `0xD3C34C3D`, because `stoi` can parse every character. The `string` in the next test, `42six`, contains the unparsable characters `six`. When you invoke `stoul` this time, the `result` is `42` and `last_character` equals `2`, the position of `s` in `six` ❹. Finally, you use `stod` to convert the `string` `2.7182818` to a `double` ❺.

**NOTE** *Boost's Lexical Cast provides an alternative, template-based approach to numeric conversions. Refer to the documentation for* `boost::lexical_cast` *available in the* `<boost/lexical_cast.hpp>` *header.*

## String View

A *string view* is an object that represents a constant, contiguous sequence of characters. It's very similar to a `const string` reference. In fact, string view classes are often implemented as a pointer to a character sequence and a length.

The STL offers the class template `std::basic_string_view` in the `<string_view>` header, which is analogous to `std::basic_string`. The template `std::basic_string_view` has a specialization for each of the four commonly used character types:

- `char` has `string_view`
- `wchar_t` has `wstring_view`
- `char16_t` has `u16string_view`
- `char32_t` has `u32string_view`

This section discusses the `string_view` specialization for demonstration purposes, but the discussion generalizes to the other three specializations.

The `string_view` class supports most of the same methods as `string`; in fact, it's designed to be a drop-in replacement for a `const string&`.

### Constructing

The `string_view` class supports default construction, so it has zero length and points to `nullptr`. Importantly, `string_view` supports implicit construction from a `const string&` or a C-style string. You can construct `string_view` from a `char*` and a `size_t`, so you can manually specify the desired length in case you want a substring or you have embedded nulls. Listing 15-19 illustrates the use of `string_view`.

```
TEST_CASE("std::string_view supports") {
  SECTION("default construction") {
    std::string_view view; ❶
    REQUIRE(view.data() == nullptr);
    REQUIRE(view.size() == 0);
    REQUIRE(view.empty());
  }
  SECTION("construction from string") {
    std::string word("sacrosanct");
    std::string_view view(word); ❷
    REQUIRE(view == "sacrosanct");
  }
  SECTION("construction from C-string") {
    auto word = "viewership";
    std::string_view view(word); ❸
    REQUIRE(view == "viewership");
  }
  SECTION("construction from C-string and length") {
    auto word = "viewership";
    std::string_view view(word, 4); ❹
    REQUIRE(view == "view");
  }
}
```

*Listing 15-19: The constructors of `string_view`*

The default-constructed `string_view` points to `nullptr` and is empty ❶. When you construct a `string_view` from a `string` ❷ or a C-style string ❸, it points to the original's contents. The final test provides the optional length argument `4`, which means the `string_view` refers to only the first four characters instead ❹.

Although `string_view` also supports copy construction and assignment, it doesn't support move construction or assignment. This design makes sense when you consider that `string_view` doesn't own the sequence to which it points.

### Supported string_view Operations

The `string_view` class supports many of the same operations as a `const string&` with identical semantics. The following lists all the shared methods between `string` and `string_view`:

**Iterators** `begin`, `end`, `rbegin`, `rend`, `cbegin`, `cend`, `crbegin`, `crend`

**Element Access** `operator[]`, `at`, `front`, `back`, `data`

**Capacity** `size`, `length`, `max_size`, `empty`

**Search** `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of`

**Extraction** `copy`, `substr`

**Comparison** `compare`, `operator==`, `operator!=` , `operator<`, `operator>`, `operator<=`, `operator>=`

In addition to these shared methods, `string_view` supports the `remove_prefix` method, which removes the given number of characters from the beginning of the `string_view`, and the `remove_suffix` method, which instead removes characters from the end. Listing 15-20 illustrates both methods.

```
TEST_CASE("std::string_view is modifiable with") {
  std::string_view view("previewing"); ❶
  SECTION("remove_prefix") {
    view.remove_prefix(3); ❷
    REQUIRE(view == "viewing");
  }
  SECTION("remove_suffix") {
    view.remove_suffix(3); ❸
    REQUIRE(view == "preview");
  }
}
```

*Listing 15-20: Modifying a `string_view` with `remove_prefix` and `remove_suffix`*

Here, you declare a `string_view` referring to the string literal `previewing` ❶. The first test invokes `remove_prefix` with `3` ❷, which removes three characters from the front of `string_view` so it now refers to `viewing`. The second test instead invokes `remove_suffix` with `3` ❸, which removes three characters from the back of the `string_view` and results in `preview`.

### Ownership, Usage, and Efficiency

Because `string_view` doesn't own the sequence to which it refers, it's up to you to ensure that the lifetime of the `string_view` is a subset of the referred-to sequence's lifetime.

Perhaps the most common usage of `string_view` is as a function parameter. When you need to interact with an immutable sequence of characters, it's the first port of call. Consider the `count_vees` function in Listing 15-21, which counts the frequency of the letter `v` in a sequence of characters.

```
#include <string_view>

size_t count_vees(std::string_view my_view❶) {
  size_t result{};
  for(auto letter : my_view) ❷
    if (letter == 'v') result++; ❸
  return result; ❹
}
```

*Listing 15-21: The count_vees function*

The `count_vees` function takes a `string_view` called `my_view` ❶, which you iterate over using a range-based `for` loop ❷. Each time a character in `my_view` equals `v`, you increment a `result` variable ❸, which you return after exhausting the sequence ❹.

You could reimplement Listing 15-21 by simply replacing `string_view` with `const string&`, as demonstrated in Listing 15-22.

```
#include <string>

size_t count_vees(const std::string& my_view) {
--snip--
}
```

*Listing 15-22: The count_vees function reimplemented to use a const string& instead of a string_view*

If `string_view` is just a drop-in replacement for a `const string&`, why bother having it? Well, if you invoke `count_vees` with a `std::string`, there's no difference: modern compilers will emit the same code.

If you instead invoke `count_vees` with a string literal, there's a big difference: when you pass a string literal for a `const string&`, you construct a `string`. When you pass a string literal for a `string_view`, you construct a `string_view`. Constructing a `string` is probably more expensive, because it might have to allocate dynamic memory and it definitely has to copy characters. A `string_view` is just a pointer and a length (no copying or allocating is required).

## Regular Expressions

A *regular expression*, also called a *regex*, is a string that defines a search pattern. Regexes have a long history in computer science and form a sort of mini-language for searching, replacing, and extracting language data. The STL offers regular expression support in the `<regex>` header.

When used judiciously, regular expressions can be tremendously powerful, declarative, and concise; however, it's also easy to write regexes that are totally inscrutable. Use regexes deliberately.

### *Patterns*

You build regular expressions using strings called *patterns*. Patterns represent a desired set of strings using a particular regular expression grammar that sets the syntax for building patterns. In other words, a pattern defines the subset of all possible strings that you're interested in. The STL supports a handful of grammars, but the focus here will be on the very basics of the default grammar, the modified ECMAScript regular expression grammar (see [re.grammar] for details).

#### Character Classes

In the ECMAScript grammar, you intermix literal characters with special markup to describe your desired strings. Perhaps the most common markup is a *character class*, which stands in for a set of possible characters: `\d` matches any digit, `\s` matches any whitespace, and `\w` matches any alphanumeric ("word") character.

Table 15-8 lists a few example regular expressions and possible interpretations.

**Table 15-8:** Regular Expression Patterns Using Only Character Classes and Literals

| Regex pattern | Possibly describes |
|---|---|
| `\d\d\d-\d\d\d-\d\d\d\d` | An American phone number, such as 202-456-1414 |
| `\d\d:\d\d \wM` | A time in HH:MM AM/PM format, such as 08:49 PM |
| `\w\w\d\d\d\d\d` | An American ZIP code including a prepended state code, such as NJ07932 |
| `\w\d-\w\d` | An astromech droid identifier, such as R2-D2 |
| `c\wt` | A three-letter word starting with c and ending with t, such as cat or cot |

You can also invert a character class by capitalizing the *d*, *s*, or *w* to give the opposite: `\D` matches any non-digit, `\S` matches any non-whitespace, and `\W` matches any non-word character.

In addition, you can build your own character classes by explicitly enumerating them between square brackets `[]`. For example, the character class `[02468]` includes even digits. You can also use hyphens as shortcuts to include implied ranges, so the character class `[0-9a-fA-F]` includes any hexadecimal digit whether the letter is capitalized or not. Finally, you can invert a custom character class by prepending the list with a caret `^`. For example, the character class `[^aeiou]` includes all non-vowel characters.

#### Quantifiers

You can save some typing by using *quantifiers*, which specify that the character directly to the left should be repeated some number of times. Table 15-9 lists the regex quantifiers.

**Table 15-9:** Regular Expression Quantifiers

| Regex quantifier | Specifies a quantity of |
|---|---|
| * | 0 or more |
| + | 1 or more |
| ? | 0 or 1 |
| {n} | Exactly n |
| {n,m} | Between n and m, inclusive |
| {n,} | At least n |

Using quantifiers, you can specify all words beginning with *c* and ending with *t* using the pattern `c\w*t`, because `\w*` matches any number of word characters.

### Groups

A *group* is a collection of characters. You can specify a group by placing it within parentheses. Groups are useful in several ways, including specifying a particular collection for eventual extraction and for quantification.

For example, you could improve the ZIP pattern in Table 15-8 to use quantifiers and groups, like this:

```
(\w{2})?❶(\d{5})❷(-\d{4})?❸
```

Now you have three groups: the optional state ❶, the ZIP code ❷, and an optional four-digit suffix ❸. As you'll see later on, these groups make parsing from regexes much easier.

### Other Special Characters

Table 15-10 lists several other special characters available for use in regex patterns.

**Table 15-10:** Example Special Characters

| Character | Specifies |
|---|---|
| X\|Y | Character X or Y |
| \Y | The special character Y as a literal (in other words, escape it) |
| \n | Newline |
| \r | Carriage return |
| \t | Tab |
| \0 | Null |
| \xYY | The hexadecimal character corresponding to YY |

## *basic_regex*

The STL's `std::basic_regex` class template in the `<regex>` header represents a regular expression constructed from a pattern. The `basic_regex` class accepts two template parameters, a character type and an optional traits class. You'll almost always want to use one of the convenience specializations: `std::regex` for `std::basic_regex<char>` or `std::wregex` for `std::basic_regex<wchar_t>`.

The primary means of constructing a `regex` is by passing a string literal containing your regex pattern. Because patterns will require a lot of escaped characters—especially the backslash `\`—it's a good idea to use raw string literals, such as `R"()"`. The constructor accepts a second, optional parameter for specifying syntax flags like the regex grammar.

Although `regex` is used primarily as input into regular expression algorithms, it does offer a few methods that users can interact with. It supports the usual copy and move construction and assignment suite and `swap`, plus the following:

- `assign(`**`s`**`)` reassigns the pattern to **`s`**
- `mark_count()` returns the number of groups in the pattern
- `flags()` returns the syntax flags issued at construction

Listing 15-23 illustrates how you could construct a ZIP code `regex` and inspect its subgroups.

```
#include <regex>

TEST_CASE("std::basic_regex constructs from a string literal") {
  std::regex zip_regex{ R"((\w{2})?(\d{5})(-\d{4})?)" }; ❶
  REQUIRE(zip_regex.mark_count() == 3); ❷
}
```

*Listing 15-23: Constructing a `regex` using a raw string literal and extracting its group count*

Here, you construct a `regex` called `zip_regex` using the pattern `(\w{2})?(\d{5})(-\d{4})?` ❶. Using the `mark_count` method, you see that `zip_regex` contains three groups ❷.

## *Algorithms*

The `<regex>` class contains three algorithms for applying `std::basic_regex` to a target string: matching, searching, or replacing. Which you choose depends on the task at hand.

### Matching

*Matching* attempts to marry a regular expression to an *entire* `string`. The STL provides the `std::regex_match` function for matching, which has four overloads.

First, you can provide `regex_match` a `string`, a C-string, or a begin and end iterator forming a half-open range. The next parameter is an optional reference to a std::match_results object that receives details about the match. The next parameter is a std::basic_regex that defines the matching, and the final parameter is an optional std::regex_constants::match_flag_type that specifies additional matching options for advanced use cases. The regex_match function returns a bool, which is true if it found a match; otherwise, it's false.

To summarize, you can invoke regex_match in the following ways:

```
regex_match(beg, end, [mr], rgx, [flg])
regex_match(str, [mr], rgx, [flg])
```

Either provide a half-open range from beg to end or a string/C-string str to search. Optionally, you can provide a match_results called mr to store all the details of any matches found. You obviously have to provide a regex rgx. Finally, the flags flg are seldom used.

**NOTE** *For details on match flags* **flg***, refer to [re.alg.match].*

A *submatch* is a subsequence of the matched string that corresponds to a group. The ZIP code–matching regular expression (\w{2})(\d{5})(-\d{4})? can produce two or three submatches depending on the string. For example, TX78209 contains the two submatches TX and 78209, and NJ07936-3173 contains the three submatches NJ, 07936, and -3173.

The match_results class stores zero or more std::sub_match instances. A sub_match is a simple class template that exposes a length method to return the length of a submatch and a str method to build a string from the sub_match.

Somewhat confusingly, if regex_match successfully matches a string, match_results stores the entire matched string as its first element and then stores any submatches as subsequent elements.

The match_results class provides the operations listed in Table 15-11.

**Table 15-11:** Supported Operations of match_results

| Operation | Description |
|---|---|
| **mr**.empty() | Checks whether the match was successful. |
| **mr**.size() | Returns the number of submatches. |
| **mr**.max_size() | Returns the maximum number of submatches. |
| **mr**.length([**i**]) | Returns the length of the submatch i, which defaults to 0. |
| **mr**.position([**i**]) | Returns the character of the first position of submatch i, which defaults to 0. |
| **mr**.str([**i**]) | Returns the string representing submatch i, which defaults to 0. |
| **mr**[**i**] | Returns a reference to a std::sub_match class corresponding to submatch i, which defaults to 0. |
| **mr**.prefix() | Returns a reference to a std::sub_match class corresponding to the sequence before the match. |

*(continued)*

**Table 15-11:** Supported Operations of match_results (continued)

| Operation | Description |
|---|---|
| `mr.suffix()` | Returns a reference to a `std::sub_match` class corresponding to the sequence after the match. |
| `mr.format(str)` | Returns a `string` with contents according to the format string **str**. There are three special sequences: **$'** for the characters before a match, **$'** for the characters after the match, and **$&** for the matched characters. |
| `mr.begin()`<br>`mr.end()`<br>`mr.cbegin()`<br>`mr.cend()` | Returns the corresponding iterator to the sequence of submatches. |

The `std::sub_match` class template has predefined specializations to work with common string types:

- `std::csub_match` for a `const char*`
- `std::wcsub_match` for a `const wchar_t*`
- `std::ssub_match` for a `std::string`
- `std::wssub_match` for a `std::wstring`

Unfortunately, you'll have to keep track of all these specializations manually due to the design of `std::regex_match`. This design generally befuddles newcomers, so let's look at an example. Listing 15-24 uses the ZIP code regular expression `(\w{2})(\d{5})(-\d{4})?` to match against the strings `NJ07936-3173` and `Iomega Zip 100`.

```
#include <regex>
#include <string>

TEST_CASE("std::sub_match") {
  std::regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" }; ❶
  std::smatch results; ❷
  SECTION("returns true given matching string") {
    std::string zip("NJ07936-3173");
    const auto matched = std::regex_match(zip, results, regex); ❸
    REQUIRE(matched); ❹
    REQUIRE(results[0] == "NJ07936-3173"); ❺
    REQUIRE(results[1] == "NJ"); ❻
    REQUIRE(results[2] == "07936");
    REQUIRE(results[3] == "-3173");
  }
  SECTION("returns false given non-matching string") {
    std::string zip("Iomega Zip 100");
    const auto matched = std::regex_match(zip, results, regex); ❼
    REQUIRE_FALSE(matched); ❽
    }
}
```

*Listing 15-24: A `regex_match` attempts to match a `regex` to a `string`.*

You construct a regex with the raw literal `R"((\w{2})(\d{5})(-\d{4})?)"` ❶ and default construct an `smatch` ❷. In the first test, you `regex_match` the valid ZIP code `NJ07936-3173` ❸, which returns the `true` value `matched` to indicate success ❹. Because you provide an `smatch` to `regex_match`, it contains the valid ZIP code as the first element ❺, followed by each of the three subgroups ❻.

In the second test, you `regex_match` the invalid ZIP code `Iomega Zip 100` ❼, which fails to match and returns `false` ❽.

### Searching

*Searching* attempts to match a regular expression to a *part* of a string. The STL provides the `std::regex_search` function for searching, which is essentially a replacement for `regex_match` that succeeds even when only a part of a string matches a `regex`.

For example, `The string NJ07936-3173 is a ZIP Code.` contains a ZIP code. But applying the ZIP regular expression to it using `std::regex_match` will return `false` because the regex doesn't match the *entire* string. However, applying `std::regex_search` instead would yield `true` because the string embeds a valid ZIP code. Listing 15-25 illustrates `regex_match` and `regex_search`.

```
TEST_CASE("when only part of a string matches a regex, std::regex_ ") {
  std::regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" }; ❶
  std::string sentence("The string NJ07936-3173 is a ZIP Code."); ❷
  SECTION("match returns false") {
    REQUIRE_FALSE(std::regex_match(sentence, regex)); ❸
  }
  SECTION("search returns true") {
    REQUIRE(std::regex_search(sentence, regex)); ❹
  }
}
```

*Listing 15-25: Comparing `regex_match` and `regex_search`*

As before, you construct the ZIP `regex` ❶. You also construct the example string `sentence`, which embeds a valid ZIP code ❷. The first test calls `regex_match` with `sentence` and `regex`, which returns `false` ❸. The second test instead calls `regex_search` with the same arguments and returns `true` ❹.

### Replacing

*Replacing* substitutes regular expression occurrences with replacement text. The STL provides the `std::regex_replace` function for replacing.

In its most basic usage, you pass `regex_replace` three arguments:

- A source `string`/C-string/half-open range to search
- A regular expression
- A replacement string

As an example, Listing 15-26 replaces all the vowels in the phrase `queueing and cooeeing in eutopia` with underscores (`_`).

```
TEST_CASE("std::regex_replace") {
  std::regex regex{ "[aeoiu]" }; ❶
  std::string phrase("queueing and cooeeing in eutopia"); ❷
  const auto result = std::regex_replace(phrase, regex, "_"); ❸
  REQUIRE(result == "q_____ng _nd c_____ng _n __t_p__"); ❹
}
```

*Listing 15-26: Using `std::regex_replace` to substitute underscores for vowels in a `string`*

You construct a `std::regex` that contains the set of all vowels ❶ and a `string` called `phrase` containing the vowel-rich contents `queueing and cooeeing in eutopia` ❷. Next, you invoke `std::regex_replace` with `phrase`, the regex, and the string literal _ ❸, which replaces all vowels with underscores ❹.

**NOTE** *Boost Regex provides regular expression support mirroring the STL's in the `<boost/regex.hpp>` header. Another Boost library, Xpressive, offers an alternative approach with regular expressions that you can express directly in C++ code. It has some major advantages, such as expressiveness and compile-time syntax checking, but the syntax necessarily diverges from standard regular expression syntaxes like POSIX, Perl, and ECMAScript.*

## Boost String Algorithms

Boost's String Algorithms library offers a bounty of `string` manipulation functions. It contains functions for common tasks related to `string`, such as trimming, case conversion, finding/replacing, and evaluating characteristics. You can access all the Boost String Algorithms functions in the `boost::algorithm` namespace and in the `<boost/algorithm/string.hpp>` convenience header.

### *Boost Range*

*Range* is a concept (in the Chapter 6 compile-time polymorphism sense of the word) that has a beginning and an end that allow you to iterate over constituent elements. The range aims to improve the practice of passing a half-open range as a pair of iterators. By replacing the pair with a single object, you can *compose* algorithms together by using the range result of one algorithm as the input to another. For example, if you wanted to transform a range of strings to all uppercase and sort them, you could pass the results of one operation directly into the other. This is not generally possible to do with iterators alone.

Ranges are not currently part of the C++ standard, but several experimental implementations exist. One such implementation is Boost Range, and because Boost String Algorithms uses Boost Range extensively, let's look at it now.

The Boost Range concept is like the STL container concept. It provides the usual complement of `begin`/`end` methods to expose iterators over the elements in the range. Each range has a *traversal category*, which indicates the range's supported operations:

- A *single-pass range* allows one-time, forward iteration.
- A *forward range* allows (unlimited) forward iteration and satisfies single-pass range.
- A *bidirectional range* allows forward and backward iteration and satisfies forward range.
- A *random-access range* allows arbitrary element access and satisfies bidirectional range.

Boost String Algorithms is designed for `std::string`, which satisfies the random-access range concept. For the most part, the fact that Boost String Algorithms accepts Boost Range rather than `std::string` is a totally transparent abstraction to users. When reading the documentation, you can mentally substitute `Range` with `string`.

### Predicates

Boost String Algorithms incorporates predicates extensively. You can use them directly by bringing in the `<boost/algorithm/string/predicate.hpp>` header. Most of the predicates contained in this header accept two ranges, `r1` and `r2`, and return a `bool` based on their relationship. The predicate `starts_with`, for example, returns `true` if `r1` begins with `r2`.

Each predicate has a case-insensitive version, which you can use by prepending the letter `i` to the method name, such as `istarts_with`. Listing 15-27 illustrates `starts_with` and `istarts_with`.

```
#include <string>
#include <boost/algorithm/string/predicate.hpp>

TEST_CASE("boost::algorithm") {
  using namespace boost::algorithm;
  using namespace std::literals::string_literals;
  std::string word("cymotrichous"); ❶
  SECTION("starts_with tests a string's beginning") {
    REQUIRE(starts_with(word, "cymo"s)); ❷
  }
  SECTION("istarts_with is case insensitive") {
    REQUIRE(istarts_with(word, "cYmO"s)); ❸
  }
}
```

*Listing 15-27: Both `starts_with` and `istarts_with` check a range's beginning characters.*

You initialize a `string` containing `cymotrichous` ❶. The first test shows that `starts_with` returns `true` when with `word` and `cymo` ❷. The case-insensitive version `istarts_with` also returns `true` given `word` and `cYmO` ❸.

Note that <boost/algorithm/string/predicate.hpp> also contains an all predicate, which accepts a single range r and a predicate p. It returns true if p evaluates to true for all elements of r, as Listing 15-28 illustrates.

```
TEST_CASE("boost::algorithm::all evaluates a predicate for all elements") {
  using namespace boost::algorithm;
  std::string word("juju"); ❶
  REQUIRE(all(word❷, [](auto c) { return c == 'j' || c =='u'; }❸));
}
```

*Listing 15-28: The* all *predicate evaluates if all elements in a range satisfy a predicate.*

You initialize a string containing juju ❶, which you pass to all as the range ❷. You pass a lambda predicate, which returns true for the letters j and u ❸. Because juju contains only these letters, all returns true.

Table 15-12 lists the predicates available in <boost/algorithm/string/predicate.hpp>.In this table, r, r1, and r2 are string ranges, and p is an element comparison predicate.

**Table 15-12:** Predicates in the Boost String Algorithms Library

| Predicate | Returns true if |
|---|---|
| starts_with(**r1**, **r2**, [**p**])<br>istarts_with(**r1**, **r2**) | **r1** starts with **r2**; **p** used for character-wise comparison. |
| ends_with(**r1**, **r2**, [**p**])<br>iends_with(**r1**, **r2**) | **r1** ends with **r2**; **p** used for character-wise comparison. |
| contains(**r1**, **r2**, [**p**])<br>icontains(**r1**, **r2**) | **r1** contains **r2**; **p** used for character-wise comparison. |
| equals(**r1**, **r2**, [**p**])<br>iequals(**r1**, **r2**) | **r1** equals **r2**; **p** used for character-wise comparison. |
| lexicographical_compare(**r1**, **r2**, [**p**])<br>ilexicographical_compare(**r1**, **r2**) | **r1** lexicographically less than **r2**; **p** used for character-wise comparison. |
| all(**r**, [**p**]) | All elements of **r** return true for **p**. |

Function permutations beginning with i are case-insensitive.

## Classifiers

*Classifiers* are predicates that evaluate some characteristics about a character. The <boost/algorithm/string/classification.hpp> header offers generators for creating classifiers. A *generator* is a non-member function that acts like a constructor. Some generators accept arguments for customizing the classifier.

**NOTE** *Of course, you can create your own predicates just as easily with your own function objects, like lambdas, but Boost provides a menu of premade classifiers for convenience.*

The is_alnum generator, for example, creates a classifier that determines whether a character is alphanumeric. Listing 15-29 illustrates how to use this classifier independently or in conjunction with all.

```
#include <boost/algorithm/string/classification.hpp>

TEST_CASE("boost::algorithm::is_alnum") {
  using namespace boost::algorithm;
  const auto classifier = is_alnum(); ❶
  SECTION("evaluates alphanumeric characters") {
    REQUIRE(classifier('a')); ❷
    REQUIRE_FALSE(classifier('$')); ❸
  }
  SECTION("works with all") {
    REQUIRE(all("nostarch", classifier)); ❹
    REQUIRE_FALSE(all("@nostarch", classifier)); ❺
  }
}
```

*Listing 15-29: The is_alum generator determines whether a character is alphanumeric.*

Here, you construct a classifier from the is_alnum generator ❶. The first test uses the classifier to evaluate that a is alphanumeric ❷ and $ is not ❸. Because all classifiers are predicates that operate on characters, you can use them in conjunction with the all predicate discussed in the previous section to determine that nostarch contains all alphanumeric characters ❹ and @nostarch doesn't ❺.

Table 15-13 lists the character classifications available in <boost/algorithm/string/classification.hpp>. In this table, r is a string range, and beg and end are element comparison predicates.

**Table 15-13:** Character Predicates in the Boost String Algorithms Library

| Predicate | Returns true if element is . . . |
|---|---|
| is_space | A space |
| is_alnum | An alphanumeric character |
| is_alpha | An alphabetical character |
| is_cntrl | A control character |
| is_digit | A decimal digit |
| is_graph | A graphical character |
| is_lower | A lowercase character |
| is_print | A printable character |
| is_punct | A punctuation character |
| is_upper | An uppercase character |
| is_xdigit | A hexadecimal digit |
| is_any_of(**r**) | Contained in **r** |
| is_from_range(**beg**, **end**) | Contained in the half-open range from **beg** to **end** |