

## Repetitive Searches

There are only two or three human stories,  
and they go on repeating themselves as  
fiercely as if they had never happened before.

*O Pioneers!*  
WILLA CATHER

**D**id you spot the problem with the example program that searched for code snippets in a text file at the beginning of Chapter 18? In lines that have multiple code snippets, everything between the first "<CODE>" and the last "</CODE>" is listed as a single snippet. To separate multiple snippets, we first have to change the regular expression a bit so that it doesn't swallow multiple snippets. In this case, we can replace the ".\*" with a nongreedy repetition:

```
string expr = "<CODE>(.*?)</CODE>";
```

Now, to resume searching after the text that matched, we have to change the code. To do that, instead of searching the entire line from the file, we use a pair of iterators that point at the contents of the line. After a match, we advance the first iterator to point at the character immediately following the match and search again.

```
#include <regex>
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
using std::tr1::regex; using std::tr1::regex_search;
using std::tr1::smatch;
using std::string; using std::ifstream; using std::cout;

static void show_matches(const char *fname)
{ // scan file named by fname, line by line
  ifstream input(fname);
  string str;
  smatch match;
  string expr = "<CODE>(.*?)</CODE>";
```

```

regex rgx(expr, regex::icase);
while (getline(input, str))
{ // check line for match
string::const_iterator first = str.begin();
string::const_iterator second = str.end();
while (regex_search(first, second, match, rgx))
{ // show match, then skip past it
cout << match[1] << '\n';
first += match.position() + match.length();
}
}

int main(int argc, char *argv[])
{ // search for code snippets in text file
if (argc != 2)
{ // wrong number of arguments
cout << "Usage: snippets <filename>\n";
return EXIT_FAILURE;
}
try
{ // search the file
show_matches(argv[1]);
}
catch(...)
{ // something went wrong
cout << "Error\n";
return EXIT_FAILURE;
}
return 0;
}

```

**Example 19.1** Repeated Searches  
(`regexiter/repeated.cpp`)

Don't be fooled, though: Repetitive searches aren't usually that easy to write. For example, if the regular expression begins with a "^", simply restarting the search after a match, as the previous example does, can lead to wrong answers. The following program searches the target text "abcdef" for subsequences that match the regular expression "(abc|def)". The only one is the initial "abc", but the program finds two, reporting that "def" also matches.

```

#include <regex>
#include <iostream>

```

```

#include <string>
using std::tr1::regex; using std::tr1::regex_search;
using std::tr1::smatch;
using std::string; using std::cout;

int main()
{ // search for regular expression in text
  string str = "abcdef";
  string::const_iterator first = str.begin();
  string::const_iterator second = str.end();
  smatch match;
  string expr = "^(abc|def)";
  regex rgx(expr);
  while (regex_search(first, second, match, rgx))
    { // check range for match
      cout << match[0] << '\n';
      first += match.position() + match.length();
    }
  return 0;
}

```

**Example 19.2** Naive Search Doesn't Work  
(`regexiter/naive.cpp`)

In this chapter, we look first at the complications that any repetitive search has to allow for and the techniques for fixing problems (Section 19.1). Then we look at prewritten solutions, in the form of the class template `regex_iterator` (Section 19.2) and the class template `regex_token_iterator` (Section 19.3).

## 19.1 Brute-Force Searches

In Chapter 17 we looked at several flags that you can pass to the regular expression search functions to change the details of regular expression matching. Here, we look at some of those flags again but in the context of specific problems that arise in repetitive searches. Eventually, we'll build a search function that avoids these problems; you can judge for yourself whether that's a better approach than using the two forms of regular expression iterator that the TR1 library provides.

## 19.1.1 Lost Anchors

Earlier in this chapter, we looked at a naive search function that reported two matches when applying the regular expression `"^(abc|def)"` to the target text `"abcdef"`. The problem with simply repeating the same search at a new location in the target text, as that program did, is that on the second call to `regex_search`, the target text is passed, effectively, as `"def"`, which does match the regular expression. That is, we chopped off the start of the target text but didn't tell the search function that we had done that, so it matched the `"^"` at the beginning of the regular expression to the beginning of the text that we passed, even though the text was not the beginning of the target text. The solution to this problem is simply to tell the search function that we're not at the beginning of a line, so `"^"` shouldn't match. To do that, we use the flag `match_not_bol` for all searches except the first.

```
#include <regex>
#include <iostream>
using std::tr1::regex; using std::tr1::cmatch;
using std::tr1::regex_search;
using namespace std::tr1::regex_constants;
using std::cout;

static void search(const char *tgt, const char *expr)
{ // show all subsequences of tgt that match expr
  regex rgx(expr);
  cmatch match;
  match_flag_type flags = match_default;
  const char *first = tgt;
  const char *last = tgt + strlen(tgt);
  for (;;)
  { // keep trying
    if (regex_search(first, last, match, rgx, flags))
    { // show match, move past it
      cout << match.str()
            << " at offset "
            << (match[0].first - tgt) << '\n';
      first += match.position() + match.length();
      flags = flags | match_not_bol;
    }
    else
      break;
  }
}

int main()
```

```

{ // demonstrate use of match_not_bol
  const char *expr = "^(abc|def)";
  const char *tgt = "abcdef";
  search(tgt, expr);
  return 0;
}

```

**Example 19.3** Preserving Anchors  
(`regexiter/search1.cpp`)

### 19.1.2 Lost Word Boundaries

The regular expression `"\babc"` should match the target text `"abcabc"` in one place: the first occurrence of the character sequence `"abc"`. The second `"abc"` doesn't match, because it doesn't start at a word boundary. If you try the previous `search` function with this regular expression and target text, it will find two matches. The problem is similar to the one with lost anchors: When we restart the search after the first match, the regular expression engine treats the start of the text as a word boundary. You might be tempted to fix that with the same approach we used before, by adding the flag `match_not_bow` after a successful match. But the two cases are different: A `"^"` can match only at the beginning of the original target text, so it's okay to simply disallow that match once we've moved away from the beginning of the text. A word boundary can occur inside the target text as well as at the beginning, so we have to be careful to disable matching the beginning of a word only when we're not at the beginning of a word. That can be done by checking whether the last character in a match can be in a word and, if so, prohibiting matching the beginning of a word on the next pass. That solves half the problem.

The other half of the problem occurs with a regular expression like `"\b3"`, when matched against the target text `"33"`. The first `"3"` is at a word boundary, so it should match. The second `"3"` is not at a word boundary, so it should not match. But the previous version of `search` will find that the second one matches because in the target text that's passed for the second search, it is at the beginning of the target text. So we also need to disable matching of the end of a word when the previous character cannot be in a word.

But there's an easier way. The regular expression engine already knows how to identify characters that can be in a word, so we don't need to write that logic ourselves. All we need to do is tell the engine that it can look at the character in front of the target text to decide whether it's at the beginning of a word. That's what the flag `match_prev_avail` does. Of course, we should do that only when we know that a valid character is in front of the target

text. Once we've moved forward in the target text, we know that we can look behind the current position.

```

#include <regex>
#include <iostream>
using std::tr1::regex; using std::tr1::cmatch;
using std::tr1::regex_search;
using namespace std::tr1::regex_constants;
using std::cout;

static void search(const char *tgt, const char *expr)
{ // show all subsequences of tgt that match expr
  regex rgx(expr);
  cmatch match;
  match_flag_type flags = match_default;
  const char *first = tgt;
  const char *last = tgt + strlen(tgt);
  for (;;)
  { // keep trying
    if (regex_search(first, last, match, rgx, flags))
    { // show match, move past it
      cout << match.str()
            << " at offset "
            << (match[0].first - tgt) << '\n';
      first += match.position() + match.length();
      flags = flags | match_not_bol | match_prev_avail;
    }
    else
      break;
  }
}

int main()
{ // demonstrate use of match_not_bol
  const char *expr = "\\babc";
  const char *tgt = "abcabc";
  search(tgt, expr);
  return 0;
}

```

**Example 19.4** Preserving Word Boundaries  
(`regexiter/search2.cpp`)

### 19.1.3 Empty Matches

To understand the problem that empty matches pose, we first need to look at empty matches in more detail. The regular expression "a\*" matches a sequence of zero or more repetitions of the character 'a'. When it matches zero characters, that's an empty match. If you call `regex_search` to see whether that regular expression matches the target text "bcd", the answer will be that it matches, right at the beginning.

```
#include <regex>
#include <iostream>
using std::tr1::regex; using std::tr1::cmatch;
using std::tr1::regex_search;
using namespace std::tr1::regex_constants;
using std::cout;

int main()
{ // show empty match
  const char *expr = "a*";
  regex rgx(expr);
  cmatch match;
  const char *tgt = "bcd";
  if (regex_search(tgt, match, rgx))
  { // show the match
    cout << "Matched at offset " << match.position()
          << ", with length " << match.length() << '\n';
  }
  return 0;
}
```

**Example 19.5** Empty Match  
(`regexiter/empty.cpp`)

If we use the `search` function that we wrote to eliminate lost anchors to search for all occurrences of "a\*" in the target text "bcd", we'll get into trouble. The first match is at offset 0, and its length is 0, so the function will adjust the position in the target string by zero characters and call `regex_search` again. This will loop until you get bored and terminate the program.

There are two obvious solutions. First, move the position in the target text forward by one character when you get an empty match. Second, temporarily

prohibit empty matches. Both work for some cases, but, as we'll see, you really need a combination of the two.

This version of `search` implements the first fix.

```
#include <regex>
#include <iostream>
#include <string>
using std::tr1::regex; using std::tr1::cmatch;
using std::tr1::regex_search;
using namespace std::tr1::regex_constants;
using std::cout; using std::string;

static void search(const char *tgt, const char *expr)
{ // show all subsequences of tgt that match expr
  regex rgx(expr);
  cmatch match;
  match_flag_type flags = match_default;
  const char *first = tgt;
  const char *last = tgt + strlen(tgt);
  string empty("[empty]");
  for (;;)
  { // keep trying
    if (regex_search(first, last, match, rgx, flags))
    { // show match, move past it
      cout << (match.length() ? match.str() : empty)
        << " at offset "
        << (match[0].first - tgt) << '\n';
      if (match.length() != 0)
        first += match.position() + match.length();
      else if (first == last)
        break;
      else
        ++first;
      flags = flags | match_not_bol | match_prev_avail;
    }
    else
      break;
  }
}

int main()
{ // demonstrate use of match_not_null
  const char *expr = "a*";
  const char *tgt = "bcd";
```

```

search(tgt, expr);
return 0;
}

```

**Example 19.6** Jumping Past Empty Matches  
(`regxiter/search3.cpp`)

Note the test for `first == last`; without this, the function will increment `first` past the end of the target text if an empty match occurs at the end of the target text. This works fine for the regular expression `"a*"`, but try it with the regular expression `"a*|c"`. It doesn't see that the regular expression matches the `"c"` in the target text. That's because it finds the empty match at that position and jumps past it.

This version of `search` implements the second fix, using the flag `match_not_null` to prevent empty matches until after the next successful match.

```

#include <regex>
#include <iostream>
#include <string>
using std::tr1::regex; using std::tr1::cmatch;
using std::tr1::regex_search;
using namespace std::tr1::regex_constants;
using std::cout; using std::string;

static void search(const char *tgt, const char *expr)
{ // show all subsequences of tgt that match expr
  regex rgx(expr);
  cmatch match;
  match_flag_type flags = match_default;
  match_flag_type mod = match_default;
  const char *first = tgt;
  const char *last = tgt + strlen(tgt);
  string empty("[empty]");
  for (;;)
  { // keep trying
    if (regex_search(first, last, match,
                    rgx, flags | mod))
    { // show match, move past it
      cout << (match.length() ? match.str() : empty)
            << " at offset "
            << (match[0].first - tgt) << '\n';
      if (match.length() != 0)
      { // move past match, clear modifier flags
        first += match.position() + match.length();

```

```

        mod = match_default;
    }
    else
        mod = match_not_null;
    flags = flags | match_not_bol | match_prev_avail;
    }
    else
        break;
    }
}

int main()
{ // demonstrate use of match_not_bol
  const char *expr = "a*c";
  const char *tgt = "bcd";
  search(tgt, expr);
  return 0;
}

```

**Example 19.7** Blocking Empty Matches  
(`regxiter/search4.cpp`)

This program does, indeed, find the match of "c", but it's not right, because it misses the empty match before "c". We've shut off empty matches for too long. The fix is to shut off empty matches only at the current position in the target text. To do that, we need two changes. First, we need to add the flag `match_continuous`, so that the regular expression search engine won't look for matches that occur after the start of the target text. That way, we control when the search advances further into the target text. Second, if that constrained search fails, we need to turn off the constraint and move to the next position in the target text. That is, we need to combine the two previous attempted solutions.

```

#include <regex>
#include <iostream>
#include <string>
using std::tr1::regex; using std::tr1::cmatch;
using std::tr1::regex_search;
using namespace std::tr1::regex_constants;
using std::cout; using std::string;

static void search(const char *tgt, const char *expr)
{ // show all subsequences of tgt that match expr
  regex rgx(expr);

```

```

cmatch match;
match_flag_type flags = match_default;
match_flag_type mod = match_default;
const char *first = tgt;
const char *last = tgt + strlen(tgt);
string empty("[empty]");
for (;;)
{ // keep trying
  if (regex_search(first, last, match,
    rgx, flags | mod))
  { // show match, move past it
    cout << (match.length() ? match.str() : empty)
      << " at offset "
      << (match[0].first - tgt) << '\n';
    if (match.length() != 0)
    { // move past match, clear modifier flags
      first += match.position() + match.length();
      mod = match_default;
    }
    else
      mod = match_not_null | match_continuous;
    flags = flags | match_not_bol | match_prev_avail;
  }
  else if (mod != match_default && first != last)
  { // move past failed match, clear modifier flags
    ++first;
    mod = match_default;
  }
  else
    break;
}
}

int main()
{ // demonstrate use of match_not_bol
  const char *expr = "a*|c";
  const char *tgt = "bcd";
  search(tgt, expr);
  return 0;
}

```

**Example 19.8** Fixing an Empty Match  
(regexiter/search5.cpp)

Now we have a robust search function. It's a little difficult to reuse,<sup>1</sup> though, because the action that it performs when it finds a match is embedded in the code that finds the match. Although this code can be made more generic, in most cases, you should use one of the two forms of iterator that the TR1 library provides, rather than trying to adapt this explicit loop.

## 19.2 The `regex_iterator` Class Template

The class template `regex_iterator` is defined in the header `<regex>`.

```
namespace std { // C++ standard library
  namespace tr1 { // TR1 additions

    // CLASS TEMPLATE regex_iterator
    template<class BidIt,
             class Elem = typename iterator_traits<BidIt>::value_type,
             class RXtraits = regex_traits<Elem> > class regex_iterator;

    typedef regex_iterator<const char*>
             cregex_iterator;
    typedef regex_iterator<const wchar_t*>
             wcregex_iterator;
    typedef regex_iterator<string::const_iterator>
             sregex_iterator;
    typedef regex_iterator<wstring::const_iterator>
             wsregex_iterator;

  } }

```

This class template hides the details that we looked at in the first section. A search program similar to the last example but using `regex_iterator` looks like this.

```
#include <regex>
#include <iostream>
#include <string>
#include <iterator>
#include <algorithm>
using std::tr1::regex; using std::tr1::cregex_iterator;
using std::tr1::cmatch;
using std::cout; using std::string;

```

<sup>1</sup>That is, unless “reuse” means “cut and paste,” as is often the case, for example, in Java.

```

using std::ostream_iterator; using std::copy;

namespace std { // add inserter to namespace std
template <class Elem, class Alloc>
basic_ostream<Elem, Alloc>& operator<<((
    basic_ostream<Elem, Alloc>& out, const cmatch& val)
    { // insert cmatch object into stream
    static string empty("[empty]");
    return out << (val.length() ? val.str() : empty);
    }
}

int main()
{ // demonstrate use of cregex_iterator
const char *expr = "a*|c";
const char *tgt = "bcd";
regex rgx(expr);
const char *end = tgt + strlen(tgt);
cregex_iterator first(tgt, end, rgx), last;
ostream_iterator<cmatch> out(cout, "\n");
copy(first, last, out);
return 0;
}

```

**Example 19.9** Searching  
(`regexiter/rgxiterator.cpp`)

The program creates a regular expression object, `rgx`, that holds the regular expression to search for. Then the program creates a `regex_iterator` object,<sup>2</sup> `first`, passing two iterators that delineate the target text and passing the regular expression object. The program also creates an end-of-sequence iterator, `last`. These two iterators describe a sequence of `match_results` objects, with successive elements in the sequence holding the results of successive repetitive searches. The program then creates an `ostream_iterator<cmatch>` object, which inserts `cmatch` objects into its target stream, using the `operator<<` that the program defined earlier, and passes all three iterators to the standard `copy` algorithm, which copies the contents of the range defined by `[first,last)` into the target, `out`. The tricky code that we had to write in the loop in the previous example is all handled in the `regex_iterator`'s

<sup>2</sup>The type `cregex_iterator` is a `regex_iterator` that looks at sequences delineated by `char*s`.

`operator++`, which is called inside `copy`.

```

template<class BidIt ,
        class Elem =
            typename iterator_traits<BidIt>::value_type ,
        class RXtraits = regex_traits<Elem> >
class regex_iterator {
public:
    // NESTED TYPES
    typedef basic_regex<Elem, RXtraits> regex_type;
    typedef match_results<BidIt> value_type;
    typedef std::forward_iterator_tag iterator_category;
    typedef std::ptrdiff_t difference_type;
    typedef const match_results<BidIt>* pointer;
    typedef const match_results<BidIt>& reference;

    // CONSTRUCTING AND ASSIGNING
    regex_iterator ();
    regex_iterator (BidIt, BidIt, const regex_type& re,
        regex_constants::match_flag_type =
            regex_constants::match_default);
    regex_iterator (const regex_iterator&);
    regex_iterator& operator=(const regex_iterator&);

    // DEREFERENCING
    const match_results<BidIt>& operator* ();
    const match_results<BidIt>* operator-> ();

    // MODIFYING
    regex_iterator& operator++ ();
    regex_iterator operator++ (int);

    // COMPARING
    bool operator==(const regex_iterator&) const;
    bool operator!=(const regex_iterator&) const;

private:
    // exposition only:
    BidIt first, last;
    const regex_type *pre;
    match_flag_type flags;
    match_results<BidIt> match;
};

```

The class template describes an object that can serve as a forward iterator

for an unmodifiable sequence of character sequences that match a regular expression.

The template argument `BidIt` must be a bidirectional iterator. It names the type of the iterator that will designate the target character sequence when an iterator object is created. The template arguments `Elem` and `RXtraits` name the character type and the traits type, respectively, for the regular expression type, `basic_regex<Elem, RXtraits>`, that will be passed to a `regex_iterator` object's constructor. By default, these arguments are derived from the first type argument, `BidIt`.

You create a `regex_iterator` object by passing two iterators that delineate a character range to be searched and a `basic_regex` object that holds the regular expression to search for. The resulting object points at the first matching subsequence in the target sequence. Each application of `operator++` advances the iterator to point at the next matching subsequence, until there are no more matching subsequences. At that point, the iterator compares equal to the end-of-sequence iterator, which is created with the default constructor.

The template defines several nested types (Section 19.2.1) and provides three constructors and an assignment operator (Section 19.2.2). An object can be dereferenced with `operator*` and `operator->` (Section 19.2.3), and can be incremented, to point at the next element in the output sequence, with `operator++` (Section 19.2.4). Two `regex_iterator` objects of the same type can be compared for equality (Section 19.2.5). Four predefined types for the most commonly used character types are described in Section 19.2.6.

The definition of this template includes several members marked as `exposition only`. These members are used in the descriptions of some of this template's member functions that follow. Keep in mind that these members aren't required by TR1. The rule is that the member functions have to act as if they were implemented according to the descriptions.

### 19.2.1 Nested Types

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

The type is a synonym for `basic_regex<Elem, RXtraits>`.

The typedef names the type of the regular expression object that will be used in searches. In most cases the regular expression object traffics in the same element type as the target text, so `Elem` is simply the value type of the

bidirectional iterator type `BidIt`. For example, if the target text to be searched is going to be designated by a `const char*`, the regular expression object will ordinarily have type `basic_regex<char, regex_traits<char> >`. This typedef is especially handy if you prefer qualified id's over using declarations.

```
#include <regex>
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
#include <string>

typedef std::string::const_iterator seq_t;
typedef std::tr1::regex_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;
typedef std::tr1::match_results<seq_t> match_t;

namespace std { // add inserter to namespace std
template <class Elem, class Alloc>
std::basic_ostream<Elem, Alloc>& operator<<<(
    std::basic_ostream<Elem, Alloc>& out,
    const match_t& val)
{ // insert cmatch object into stream
    static std::string empty("[empty]");
    return out << (val.length() ? val.str() : empty);
}
}

int main()
{ // split out words from text file
    rgx_t rgx("[[:alnum:]]_#]+");
    ifstream input("typename.cpp");
    std::string str;
    while (std::getline(input, str))
    { // split out words from a line of text
        rgxiter first(str.begin(), str.end(), rgx), last;
        std::ostream_iterator<rgxiter::value_type>
            tgt(std::cout, "\n");
        std::copy(first, last, tgt);
    }
    return 0;
}
```

**Example 19.10** Nested Type Name  
(`regxiter/typename.cpp`)

```
typedef match_results<BidIt> value_type;
typedef std::forward_iterator_tag iterator_category;
typedef std::ptrdiff_t difference_type;
typedef const match_results<BidIt>* pointer;
typedef const match_results<BidIt>& reference;
```

These are the usual typedefs for an iterator type.

### 19.2.2 Constructing and Assigning

```
regex_iterator<BidIt, Elem, RXtraits>::regex_iterator();
```

The constructor constructs an end-of-sequence iterator.

```
regex_iterator<BidIt, Elem, RXtraits>::regex_iterator(
    BidIt first1, BidIt last1,
    const regex_type& re,
    regex_constants::match_flag_type flgs =
        regex_constants::match_default);
```

The constructor constructs an object with initial values `first` and `last` equal to `first1` and `last1`, respectively; `pre` equal to `&re`;<sup>3</sup> and `flags` equal to `flgs`. The constructor then calls `regex_search(first, last, match, *pre, flags)`; if that call returns `false`, it marks the object as an end-of-sequence iterator.

In other words, the constructor stores the various search parameters, then searches for the first occurrence of text matching `re` in the range of characters pointed at by `[first1, last1)`. If the search succeeds, the result is stored in the member data object `match`. If the search fails, there are no matches, and the object is marked as an end-of-sequence iterator, that is, an object that compares equal to a default-constructed object.

```
#include <regex>
#include <string>
#include <iostream>
using std::string; using std::cout;
```

<sup>3</sup>Note that the iterator holds the address of the regular expression object, not a copy. Once the regular expression object is destroyed, the iterator can no longer be used.

```

typedef string::const_iterator seq_t;
typedef std::tr1::regex_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;

int main()
{ // constructing regex_iterator objects
  rgx_t rgx("not found");
  string target("this is text");
  rgxiter first(target.begin(), target.end(), rgx);
  rgxiter last;
  if (first == last)
    cout << "regular expression not found\n";
  return 0;
}

```

**Example 19.11** End-of-Sequence  
(`regexiter/endofsequence.cpp`)

```

regex_iterator<BidIt, Elem, RXtraits>::regex_iterator(
  const regex_iterator& right);
regex_iterator&
  regex_iterator<BidIt, Elem, RXtraits>::operator=(
  const regex_iterator& right);

```

The copy constructor and the assignment operator copy their argument into `*this`. After the operation, `*this == right`.

### 19.2.3 Dereferencing

```

const match_results<BidIt>&
  regex_iterator<BidIt, Elem, RXtraits>::operator*() const;
const match_results<BidIt>*
  regex_iterator<BidIt, Elem, RXtraits>::operator->() const;

```

The behavior of a program that calls either of these member operators on an end-of-sequence iterator is undefined. Otherwise, the first member operator returns a reference to the contained object `match`, and the second member operator returns a pointer to the contained object `match`.

The contained object `match` holds the results of the most recent successful search, so you can use these operators to look at those results, just as if you had written a call to `regex_search` yourself and passed a `match_results` object.

```
#include <regex>
#include <iostream>
#include <iomanip>
#include <string>
using std::string; using std::cout; using std::setw;

typedef string::const_iterator seq_t;
typedef std::tr1::regex_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;
typedef rgxiter::value_type match_t;

static void show(const match_t& match)
{ // show contents of match_t object
  for (int idx = 0; idx < match.size(); ++idx)
  { // show match[idx]
    cout << idx << ": ";
    if (match[idx].matched)
      cout << setw(match.position(idx)) << " "
        << match.str(idx) << '\n';
    else
      cout << "[not matched]";
  }
}

int main()
{ // demonstrate regex_iterator dereferencing
  string id =
    "([[:alpha:]]+)([[:space:]]+)([[:digit:]]{2,5})";
  rgx_t model_descr(id);
  string item("Emperor 400");
  rgxiter iter(item.begin(), item.end(), model_descr);
  show(*iter); // operator*
  cout << iter->str() << '\n'; // operator->
  return 0;
}
```

**Example 19.12** Examining Search Results  
(`regexiter/result.cpp`)

## 19.2.4 Modifying

```

regex_iterator
  regex_iterator<BidIt, Elem, RXtraits>::operator++(int)
  { regex_iterator tmp(*this); ++*this; return tmp; }
regex_iterator&
  regex_iterator<BidIt, Elem, RXtraits>::operator++();

```

The first member function makes a copy of `*this`, increments `*this`, and returns the copy.

The second member function begins by constructing a local variable referred to here as `start`, initialized with the value `match[0].second`.

If `match.length() == 0` and `start == end`, it marks the object as an end-of-sequence iterator and returns `*this`.

Otherwise, if `match.length() == 0`, the operator creates a temporary object, `temp_flags`, of type `match_flag_type`, holding the value `flags | match_not_null | match_continuous`. It then calls `regex_search(start, last, match, *pre, temp_flags)`. If the call returns `true`, the operator returns `*this`. Otherwise, it increments `start` and moves to the following step.

The operator next sets `flags` to `flags | match_prev_avail` and calls `regex_search(start, last, match, *pre, flags)`. If the call returns `false`, the operator marks the object as an end-of-sequence iterator. The call returns `*this`.

Whenever a call to `regex_search` returns `true`, the operator adjusts the contents of `match` so that `match.prefix().first` is equal to the previous value of `match[0].second`; for each value of `idx` for which `match[idx].matched` is `true`, `match[idx].position()` returns the value of `distance(begin, match[idx].first)`.

You probably recognized most of this text as a description of the repetitive search algorithm we developed in Section 19.1. But, the last paragraph adds a twist: Regardless of how it got there, the prefix after a successful search is the text from the end of the previous successful match up to the current match, and all the match positions are offsets from the start of the original text sequence.

Look at how the output showing the various matches is formatted in this example, which is similar to the previous one.

```
#include <regex>
#include <iostream>
#include <iomanip>
#include <string>
using std::string; using std::cout; using std::setw;

typedef string::const_iterator seq_t;
typedef std::tr1::regex_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;
typedef rgxiter::value_type match_t;

static void show(const match_t& match)
{ // show contents of match_t object
  for (int idx = 0; idx < match.size(); ++idx)
  { // show match[idx]
    cout << idx << ": ";
    if (match[idx].matched)
      cout << setw(match.position(idx)) << " "
        << match.str(idx) << '\n';
    else
      cout << "[not matched]";
  }
}

int main()
{ // demonstrate regex_iterator dereferencing
  string id =
    "([[:alpha:]]+)([[:space:]]+)([[:digit:]]{2,5})";
  rgx_t model_descr(id);
  string item("Emperor 280, Emperor 400, Whisper 60");
  rgxiter first(item.begin(), item.end(), model_descr);
  rgxiter last;
  cout << "  " << item << '\n';
  while (first != last)
    show(*first++);
  return 0;
}
```

**Example 19.13** Incrementing  
(`regexiter/increment.cpp`)

## 19.2.5 Comparing

```
bool regex_iterator<BidIt, Elem, RXtraits>::operator==(
    const regex_iterator& right) const;
bool regex_iterator<BidIt, Elem, RXtraits>::operator!=(
    const regex_iterator& right) const
{ return !(*this == right); }
```

The first member operator returns `true` only if `*this` and `right` are both end-of-sequence iterators or if `first == right.first`, `last == right.last`, `pre == right.pre`, `flags == right.flags`, and `match == right.match`. The second member operator returns `!(*this == right)`.

This rather lengthy description says what you'd expect: If you create two `regex_iterator` objects with the same arguments or by copying one onto the other, they compare equal. If you increment two equal iterators the same number of times, they still compare equal. As long as the searches—either at construction or as part of an increment—succeed, the object does not compare equal to an end-of-sequence iterator. When a search fails, as we saw earlier, the iterator object is marked as an end-of-sequence iterator; at that point, it compares equal to any other end-of-sequence iterator.

```
#include <regex>
#include <iostream>
#include <string>
using std::tr1::regex; using std::tr1::regex_iterator;
using std::string; using std::cout;

typedef regex_iterator<string::const_iterator> iter_t;

static void show_equal(const char *title,
    const iter_t& iter0, const iter_t& iter1)
{ // show equality of iterator objects
  cout << title << "\n    "
    << (iter0 == iter1 ? "equal" : "not equal") << '\n';
}

int main()
{ // demonstrate regex_iterator comparison operators
  regex rgx0("abc"), rgx1("abc");
  string tgt0("abc"), tgt1("abc");
  iter_t iter0(tgt0.begin(), tgt0.end(), rgx0);
  iter_t iter1(tgt0.begin(), tgt0.end(), rgx1);
```

```

show_equal(
    "same range, different regular expression objects",
    iter0, iter1);
iter_t iter2(tgt0.begin() + 1, tgt0.end(), rgx0);
show_equal(
    "different range, same regular expression objects",
    iter0, iter2);
iter_t iter3, iter4;
show_equal("default constructed",
    iter3, iter4);
show_equal(
    "non-default constructed and default constructed",
    iter0, iter4);
++iter0; // move past final match
show_equal(
    "incremented to end and default constructed",
    iter0, iter4);
return 0;
}

```

**Example 19.14** Comparing  
(`regexiter/compare.cpp`)

### 19.2.6 Predefined `regex_iterator` Types

```

typedef regex_iterator<const char*>
    cregex_iterator;
typedef regex_iterator<const wchar_t*>
    wcregex_iterator;
typedef regex_iterator<string::const_iterator>
    sregex_iterator;
typedef regex_iterator<wstring::const_iterator>
    wsregex_iterator;

```

As always, there are four predefined `regex_iterator` types for text sequences held in arrays of `char` and `wchar_t` and in `basic_string` objects holding elements of type `char` and `wchar_t`.

## 19.3 The `regex_token_iterator` Class Template

The class template `regex_token_iterator` is defined in the header `<regex>`.

```

namespace std { // C++ standard library
  namespace tr1 { // TR1 additions

    // CLASS TEMPLATE regex_token_iterator
  template<class BidIt,
    class Elem = typename iterator_traits<BidIt>::value_type,
    class RXtraits = regex_traits<Elem> >
    class regex_token_iterator;

  typedef regex_token_iterator<const char*>
    cregex_token_iterator;
  typedef regex_token_iterator<const wchar_t*>
    wcregex_token_iterator;
  typedef regex_token_iterator<string::const_iterator>
    sregex_token_iterator;
  typedef regex_token_iterator<wstring::const_iterator>
    wsregex_token_iterator;

} }

```

Dereferencing a `regex_iterator` object produces a `match_results` object that represents the current match. As we saw in several earlier examples, the returned object can, in turn, be used to get at various submatches of a successful match. A `regex_token_iterator` object provides direct access to submatches. When you construct a `regex_token_iterator` object, you pass an additional set of numeric arguments that designate the desired submatches. Each time you increment the iterator, it advances to the next submatch. When it runs out of submatches, the iterator moves to the next match and starts the list of submatches over again. So the explicit loop over submatches that we used earlier can be eliminated.

```

#include <regex>
#include <iostream>
#include <string>
using std::string; using std::cout;

typedef string::const_iterator seq_t;
typedef std::tr1::regex_token_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;
typedef rgxiter::value_type match;

int main()
{ // demonstrate regex_token_iterator
  string id =

```

```

    " ([:alpha:]]+)([:space:]]+)([:digit:]]{2,5})";
    rgx_t model_descr(id);
    string item("Emperor 280, Emperor 400, Whisper 60");
    int fields[] = { 0, 1, 3 };
    rgxiter first(item.begin(), item.end(),
                 model_descr, fields);
    rgxiter last;
    cout << item << '\n';
    while (first != last)
        cout << *first++ << '\n';
    return 0;
}

```

**Example 19.15** Searching  
(`regexiter/tokiterator.cpp`)

This program is much simpler than the similar one in Section 19.2.4 but doesn't provide as much information. That's because `operator*` on a `regex_token_iterator` object returns a `sub_match` object, which points at a portion of the target text and, unlike `match_results`, does not know how far into the target text this match occurred.

```

template<class BidIt,
        class Elem =
            typename iterator_traits<BidIt>::value_type,
        class RXtraits = regex_traits<Elem> >
class regex_token_iterator {
public:
    // NESTED TYPES
    typedef basic_regex<Elem, RXtraits> regex_type;
    typedef sub_match<BidIt> value_type;
    typedef std::forward_iterator_tag iterator_category;
    typedef std::ptrdiff_t difference_type;
    typedef const sub_match<BidIt>* pointer;
    typedef const sub_match<BidIt>& reference;

    // CONSTRUCTING AND ASSIGNING
    regex_token_iterator();
    regex_token_iterator(BidIt first, BidIt last,
                        const regex_type& re, int submatch = 0,
                        regex_constants::match_flag_type flags =
                            regex_constants::match_default);
    regex_token_iterator(BidIt first, BidIt last,
                        const regex_type& re,

```

```

    const std::vector<int> submatches,
    regex_constants::match_flag_type flags =
        regex_constants::match_default);
template<std::size_t N>
regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const int (&submatches)[N],
    regex_constants::match_flag_type flags =
        regex_constants::match_default);
regex_token_iterator(const regex_token_iterator&);
regex_token_iterator& operator=(
    const regex_token_iterator&);

    // DEREFERENCING
const sub_match<BidIt>& operator*() const;
const sub_match<BidIt> *operator->() const;

    // MODIFYING
regex_token_iterator& operator++();
regex_token_iterator operator++(int);

    // COMPARING
bool operator==(const regex_token_iterator& right) const;
bool operator!=(const regex_token_iterator& right) const;

private:
    // exposition only:
typedef regex_iterator<BidIt, Elem, Rxtraits> iter;
iter pos;
std::vector<int> subs;
std::size_t N;
};

```

The class template describes an object that can serve as a forward iterator for an unmodifiable sequence of character sequences that match various parts of a regular expression.

The template argument `BidIt` must be a bidirectional iterator. It names the type of the iterator that will designate the target character sequence when an iterator object is created. The template arguments `Elem` and `Rxtraits` name the character type and the traits type, respectively, for the regular expression type, `basic_regex<Elem, Rxtraits>`, that will be passed to a `regex_token_iterator` object's constructor. By default, these arguments are derived from the first type argument, `BidIt`.

You create a `regex_token_iterator` object by passing two iterators that delineate a character range to be searched and a `basic_regex` object that holds the regular expression to search for, just as you do for a `regex_iterator` object. In addition, though, you pass one or more integer values that identify the various submatches that you want to iterate through. The constructors search for the first text subsequence that matches the regular expression. The resulting object points at the first of the designated submatches in the matching subsequence. Each application of `operator++` moves to the next submatch. If the list of submatches has been exhausted, the operator searches for the next text subsequence that matches the regular expression and points at the first of the designated submatches in the matching subsequence. If there are no more matching subsequences, the iterator compares equal to the end-of-sequence iterator, which is created with the default constructor.

The template defines several nested types (Section 19.3.1) and provides five constructors and an assignment operator (Section 19.3.2). An object can be dereferenced with `operator*` and `operator->` (Section 19.3.3) and can be incremented to point at the next element in the output sequence with `operator++` (Section 19.3.4). Two `regex_token_iterator` objects of the same type can be compared for equality (Section 19.3.5). Four predefined types for the most commonly used character types are described in Section 19.3.6.

The definition of this template includes several members marked as `exposition only`. These members are used in the descriptions that follow of some of the member functions of this template. Keep in mind that these members aren't required by TR1. The rule is that the member functions have to act as if they were implemented according to the descriptions.

The descriptions also use a couple of technical terms that are defined in TR1. A *suffix iterator* is an iterator object of type `regex_token_iterator` that points at the final sequence of characters in the target text. The *current match* is `(*pos).prefix()` if `subs[N]` is `-1`; otherwise, `(*pos)[subs[N]]`.

That last term is the key to understanding how a `regex_token_iterator` determines the sequence of submatches to return. When you construct a `regex_token_iterator` object, you pass one or more integer values, as described in Section 19.3.2. Those values, in turn, determine which submatches will be returned and in what order. A value of `-1` refers to the text beginning at the end of the previous match—or at the beginning of the text sequence when the iterator object is first constructed—and ending at the beginning of the current match. After the final, failed, search, a value of `-1` refers to the text from the end of the last successful search—or the beginning of the text sequence if no search succeeded—to the end of the text sequence. Any other value refers to the corresponding capture group. Thus, a value of `0` means the

entire matched text, a value of 1 means the first capture group, and so on. Each time you increment an iterator object, it advances to the next subgroup, as determined by those integer values. When it's gone through all those values, it moves to the next match and repeats the sequence of values.

### 19.3.1 Nested Types

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

The type is a synonym for `basic_regex<Elem, RXtraits>`.

The typedef names the type of the regular expression object that will be used in searches. For details, see the discussion in Section 19.2.1.

```
typedef basic_string<Elem> value_type;
typedef std::forward_iterator_tag iterator_category;
typedef std::ptrdiff_t difference_type;
typedef const basic_string<Elem>* pointer;
typedef const basic_string<Elem>& reference;
```

These are the usual typedefs for an iterator type.

### 19.3.2 Constructing and Assigning

```
regex_token_iterator<BidIt, Elem, RXtraits>::
  regex_token_iterator ();
```

The constructor constructs an end-of-sequence iterator.

```
regex_token_iterator<BidIt, Elem, RXtraits>::
  regex_token_iterator (
    BidIt first, BidIt last,
    const regex_type& re, int submatch = 0,
    regex_constants::match_flag_type flags =
      regex_constants::match_default);
regex_token_iterator<BidIt, Elem, RXtraits>::
  regex_token_iterator (
    BidIt first, BidIt last,
    const regex_type& re,
    const std::vector<int> submatches,
    regex_constants::match_flag_type flags =
      regex_constants::match_default);
```

```

template<std::size_t N>
regex_token_iterator<BidIt, Elem, RXtraits>::
    regex_token_iterator(
        BidIt first, BidIt last,
        const regex_type& re, const int (&submatches)[N],
        regex_constants::match_flag_type flags =
            regex_constants::match_default);

```

The first constructor stores the value of `submatch` in `subs`. The second and third constructors each copy their argument `submatch` into `subs`.

The constructors then set the value of `N` to 0 and the value of `pos` to `iter(first, last, re, flags)`. If `pos` is not an end-of-sequence iterator, the constructors set `res` to the address of the current match. Otherwise, if any of the values stored in `subs` is `-1`, the constructors set `*this` to be a suffix iterator that points at the entire text sequence `[first, last)`. Otherwise, the constructors set `*this` to an end-of-sequence iterator.

The first constructor takes exactly one integer argument, which designates the sub-group to be returned by the iterator. To see the entire matching text, pass the value 0. To see the *n*th capture group, pass *n*. To see the text that precedes the match, pass `-1`.

```

#include <regex>
#include <iostream>
#include <string>
using std::string; using std::cout;

typedef string::const_iterator seq_t;
typedef std::tr1::regex_token_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;
typedef rgxiter::value_type match;

static void show(int field)
{ // demonstrate single-field constructor
    string id =
        "([[:alpha:]]+)([[:space:]]+)([[:digit:]]{2,5})";
    rgx_t model_descr(id);
    string item("Emperor 280, Emperor 400, Whisper 60");
    rgxiter first(item.begin(), item.end(),
        model_descr, field);
    rgxiter last;
    while (first != last)

```

```

        cout << *first++ << '\n';
    }

int main()
{ // demonstrate regex_token_iterator single-field constructor
  cout << "Full match:\n";
  show(0);
  cout << "\nModel name:\n";
  show(1);
  cout << "\nModel number:\n";
  show(3);
  cout << "\nSeparators:\n";
  show(-1);
  return 0;
}

```

**Example 19.16** Viewing a Single Submatch  
(`regxiter/single.cpp`)

The second and third constructors take one or more integer arguments, either as a C++ `vector<int>` or as a C-style array of `int`.

```

#include <regex>
#include <iostream>
#include <string>
#include <vector>
using std::string; using std::cout; using std::vector;

typedef string::const_iterator seq_t;
typedef std::tr1::regex_token_iterator<seq_t> rgxiter;
typedef rgxiter::regex_type rgx_t;
typedef rgxiter::value_type match;

static void show(const vector<int>& fields)
{ // demonstrate multiple-field constructor
  string id =
    "([[:alpha:]]+)([[:space:]]+)([[:digit:]]{2,5})";
  rgx_t model_descr(id);
  string item("Emperor 280, Emperor 400, Whisper 60");
  rgxiter first(item.begin(), item.end(),
    model_descr, fields);
  rgxiter last;
  while (first != last)
    cout << *first++ << '\n';
}

```

```

int main()
{ // demonstrate regex_token_iterator multiple-field constructor
  vector<int> fields;
  fields.push_back(0);
  cout << "Full match:\n";
  show(fields);
  fields.push_back(3);
  cout << "Full match, model number:\n";
  show(fields);
  fields.push_back(1);
  cout << "Full match, model number, model name:\n";
  show(fields);
  return 0;
}

```

**Example 19.17** Viewing Multiple Submatches  
(`regexiter/multiple.cpp`)

```

regex_token_iterator<BidIt, Elem, RXtraits>::
  regex_token_iterator(
    const regex_token_iterator& right);
regex_token_iterator&
regex_token_iterator<BidIt, Elem, RXtraits>::
  operator=(
    const regex_token_iterator& right);

```

The copy constructor and assignment operator each copy their argument into `*this`. After the operation, `*this == right`.

### 19.3.3 Dereferencing

```

const basic_string<Elem>&
  regex_token_iterator<BidIt, Elem, RXtraits>::
  operator*() const;
const basic_string<Elem>*
  regex_token_iterator<BidIt, Elem, RXtraits>::
  operator->() const;

```

The behavior of a program that calls either of these member operators on an end-of-sequence iterator is undefined. Otherwise, the first member

operator returns a reference to the current match, and the second member operator returns a pointer to the current match.

### 19.3.4 Modifying

```
regex_token_iterator
  regex_token_iterator<BidIt, Elem, RXtraits>::
  operator++(int)
    {regex_token_iterator tmp(*this); ++*this; return tmp;}
regex_token_iterator&
  regex_token_iterator<BidIt, Elem, RXtraits>::operator++();
```

The first member function makes a copy of *\*this*, increments *\*this*, and returns the copy.

If the stored iterator *pos* is an end-of-sequence iterator, the second operator marks *\*this* as an end-of-sequence iterator. Otherwise, the operator increments the stored value *N*; if the result is equal to *subs.size()*, it sets the stored value *N* to 0 and increments the stored iterator *pos*. If incrementing the stored iterator leaves it unequal to an end-of-sequence iterator, the operator does nothing further. Otherwise, if the end of the preceding match was at the end of the character sequence, the operator marks *\*this* as an end-of-sequence iterator. Otherwise, the operator repeatedly increments the stored value *N* until *N == subs.size()*, in which case it marks *\*this* as an end-of-sequence iterator or until *subs[N] == -1*, thus ensuring that the next dereference will return the suffix of the last successful match. In all cases, the operator returns *\*this*.

To better understand how a submatch selector of *-1* works, think of the target text as a sequence of subsequences  $U_1M_1U_2M_2\cdots U_mM_mU_{m+1}$ , where the various subsequences  $M_i$  match the regular expression, and the various subsequences  $U_i$  do not match the regular expression. A selector of *-1* selects the  $U_i$  subsequences, including the final nonmatching subsequence  $U_{m+1}$  if it is not empty.

```
#include <regex>
#include <iostream>
#include <string>
using std::string; using std::cout;

typedef string::const_iterator seq_t;
typedef std::tr1::regex_token_iterator<seq_t> rgxiter;
```

```

typedef regexiter::regex_type rgx_t;
typedef regexiter::value_type match;

int main()
{ // demonstrate use of selector value -1
  string csv("[[:space:]]*,[[:space:]]*");
  rgx_t rgx(csv);
  string data("Ron Mars, 2114 East St.    , Biloxi, MI");
  regexiter first(data.begin(), data.end(), rgx, -1);
  regexiter last;
  while (first != last)
    cout << *first++ << '\n';
  return 0;
}

```

**Example 19.18** Selecting Separators  
(`regexiter/select.cpp`)

### 19.3.5 Comparing

```

bool regex_token_iterator<BidIt, Elem, RXtraits>::
  operator==(
    const regex_token_iterator& right) const;
bool regex_token_iterator<BidIt, Elem, RXtraits>::
  operator!=(
    const regex_token_iterator& right) const
{ return !(*this == right); }

```

The first member function returns `true` if `*this` and `right` are both end-of-sequence iterators or if both are suffix iterators that point at the same text sequence. Otherwise, if either of them is an end-of-sequence iterator or a suffix iterator, the member function returns `false`. Otherwise, the member function returns `pos == right.pos && subs == right.subs && N == right.N`.

The second member function returns `!(*this == right)`.

Two `regex_token_iterator` objects compare equal if they were constructed from the same regular expression argument and equal other arguments, and

they have been incremented the same number of times. When you make a copy of a `regex_token_iterator` object, the first requirement is obviously satisfied, so a copy of a `regex_token_iterator` object is equal to the object it was copied from if both have been incremented the same number of times since the copy was made.

```
#include <regex>
#include <iostream>
#include <string>
using std::tr1::regex;
using std::tr1::regex_token_iterator;
using std::string; using std::cout;

typedef string::const_iterator siter;
typedef regex_token_iterator<siter> iter_t;

static void show_equal(const char *title,
    const iter_t& iter0, const iter_t& iter1)
{ // show equality of iterator objects
  cout << title << "\n      "
    << (iter0 == iter1 ? "equal" : "not equal") << '\n';
}

int main()
{ // demonstrate regex_token_iterator comparison operators
  string csv("[[:space:]]*,[[:space:]]*");
  regex rgx(csv);
  string data("Ron Mars, 2114 East St.      , Biloxi, MI");
  int selector0 [] = { 0, 1 };
  int selector1 [] = { 0, 1 };
  int selector2 [] = { 1, 0 };
  iter_t iter0(data.begin(), data.end(), rgx, selector0);
  iter_t iter1(data.begin(), data.end(), rgx, selector0);
  show_equal("equal arguments", iter0, iter1);
  iter_t iter2(data.begin(), data.end(), rgx, selector1);
  show_equal("equal selectors", iter0, iter2);
  iter_t iter3(data.begin(), data.end(), rgx, selector2);
  show_equal("unequal selectors", iter0, iter3);

  iter_t iter4(++iter0);
  show_equal("copy", iter0, iter4);
  ++iter0;
  show_equal("unequal increments", iter0, iter4);
  ++iter4;
```

```

show_equal("equal increments", iter0, iter4);
return 0;
}

```

**Example 19.19** Comparing  
(`regexiter/comparetok.cpp`)

### 19.3.6 Predefined `regex_token_iterator` Types

```

typedef regex_token_iterator<const char*>
    cregex_token_iterator;
typedef regex_token_iterator<const wchar_t*>
    wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator>
    sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator>
    wsregex_token_iterator;

```

As always, there are four predefined `regex_token_iterator` types for text sequences held in arrays of `char` and `wchar_t` and in `basic_string` objects holding elements of type `char` and `wchar_t`.

## Exercises

**Exercise 1** For each of the following errors, write a simple test case containing the error, and try to compile it. In the error messages, look for the key words that relate to the error in the code.

1. Attempting to construct a `regex_iterator` object by passing a pair of iterators whose character type is different from the `regex_iterator` type's character type
2. Attempting to construct a `regex_iterator` object by passing a regular expression object whose element type or traits type is different from the `regex_iterator` type's element type or traits type
3. Attempting to construct a `regex_token_iterator` object by passing a pair of iterators whose character type is different from the `regex_token_iterator` type's character type
4. Attempting to construct a `regex_token_iterator` object by passing a regular expression object whose element type or traits type is different from the `regex_token_iterator` type's element type or traits type

5. Attempting to construct a `regex_token_iterator` object by passing a field specifier as a pointer to `int` instead of an array of `int`
6. Attempting to decrement a `regex_iterator` object
7. Attempting to decrement a `regex_token_iterator` object

**Exercise 2** In the first part of this chapter, I mentioned that it's a little hard to reuse the brute-force loop. In this exercise, we look at a couple of possible approaches to reuse and at doing the same thing with regular expression iterators.

1. Write a program that has a copy of the code of the `search` function in Example 19.8. Change the search function so that for a successful match, it shows the contents of the first capture group instead of the entire match. Now use the function to copy to `cout` all text that occurs between the tags "`<CODE>`" and "`</CODE>`"<sup>4</sup> in an HTML file of your choosing.<sup>5</sup>
2. Now write another program that has a copy of the code of the `search` function in Example 19.8. Change the search function into a template function with a template type parameter named `Fn` and an additional function call argument, `Fn func`. Also replace the code that shows the match by inserting it into `cout` with a call to `func(match)`. Now use the function for the same search as in the preceding part of this exercise.<sup>6</sup>
3. Write a program that uses a `regex_iterator` object to do the same search.
4. Write a program that uses a `regex_token_iterator` object to do the same search.
5. Now change all four programs to copy to `cout` all text that occurs between the tags "`<CODE>`" and "`</CODE>`" or between the tags "`<PRE>`" and "`</PRE>`".<sup>7</sup>

---

<sup>4</sup>That is, search for text matching the regular expression "`<CODE>(.*?)</CODE>`"; for each successful match, write out the contents of capture group 1.

<sup>5</sup>*Hint:* Read the entire text file into a string object by creating an `ifstream` object to read the file and a `basic_ostringstream` object to build the string, and inserting the buffer returned by the `ifstream`'s member function `rdbuf()` into the `basic_ostringstream` object.

<sup>6</sup>You'll have to write a callable type whose function call operator takes a `match_results` object and copies the first capture group to `cout`.

<sup>7</sup>That is, search for text matching the regular expression "`<(CODE|PRE)>(.*?)</\1>`"; for each successful match, write out the contents of capture group 2.

**Exercise 3** Use a pair of `regex_iterator` objects to search for valid hostnames<sup>8</sup> in an HTML file, and use the utility function you wrote for Exercise 2 in Chapter 18 to show the contents of each successful match.

**Exercise 4** Write a program that uses a pair of `regex_token_iterator` objects to extract data fields from a comma-separated file. Don't forget to allow for spaces and tabs before and after each comma.<sup>9</sup>

**Exercise 5** Write a program that puts the integer values 1 and 4 into a `vector<int>` and passes that vector as the field specifier in the constructor of a `regex_token_iterator` object. Use that object to search for your favorite regular expression. Now put the same values into an array of `int`, pass that array to the constructor, and repeat the search. What happens if the field index is higher than the index of the last capture group in the regular expression? What happens if you repeat a field index in the initializer?

**Exercise 6** HTML cross-references have the form `<A HREF="reference">text</A>` and `<A NAME="reference">text</A>`. The first is a link, and the second is the target of a link. In both cases, the reference is in quotes. Write a program that uses a pair of `regex_token_iterator` objects to search for cross-references in an HTML file and shows, for each cross-reference, either "HREF=" or "NAME=", as appropriate, followed by text of the reference.

---

<sup>8</sup>See Exercise 2 in Chapter 17 for a suitable regular expression.

<sup>9</sup>*Hint:* Write a regular expression that describes the separator, and use an iterator that shows the text that doesn't match the separator.