Repeated Search

We will now introduce the concept of iterators in regular expressions.

WE'LL COVER THE FOLLOWING

It's quite convenient to iterate with `std::regex_iterator` and `std::regex_token_iterator` over the matched texts. `std::regex_iterator` matched texts and their capture groups. `std::regex_token_iterator` supports more. We can address the components of each capture and by using a negative index, we can access the text between the matches.

std::regex_iterator

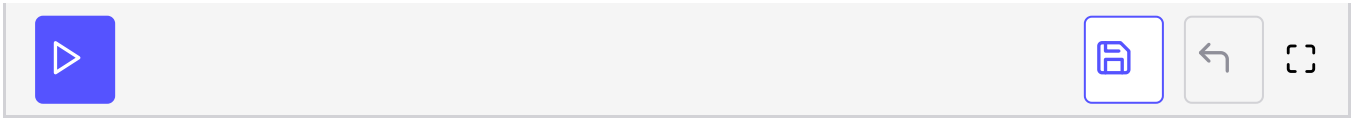
C++ defines the following four type synonyms for `std::regex_iterator`.

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

We can use `std::regex_iterator` to count the occurrences of the words in a text:

```
#include <regex>

#include <iostream>
#include <string>
#include <unordered_map>

int main(){

  std::cout << std::endl;

  // Bjarne Stroustrup about C++0x on http://www2.research.att.com/~bs/C++0xFAQ.html
```

```
  std::string text{"That's a (to me) amazingly frequent question. It may be the most frequent

  // regular expression for a word

  std::regex wordReg(R"(\w+)");

  // get all words from text
  std::sregex_iterator wordItBegin(text.begin(), text.end(), wordReg);
  const std::sregex_iterator wordItEnd;

  // use unordered_map to count the wourds
  std::unordered_map<std::string, std::size_t> allWords;

  // count the words
  for (; wordItBegin != wordItEnd;++wordItBegin){
    ++allWords[wordItBegin->str()];
  }

  for ( auto wordIt: allWords) std::cout << wordIt.first << ": " << wordIt.second << "\n" ;

  std::cout << "\n\n";

}
```



std::regex_iterator

To count how many words consist of at least one character, use `(\w+)`. This regular expression is used to define the beginning iterator, `wordItBegin`, and the end iterator `wordItEnd`. Iterating through the matches happens in the for loop. Each word increments the counter: `++allWords[wordItBegin->str()]`. A counter will be created at this point in the process and set equal to one if a counter wasn't already created in `allWords`.

std::regex_token_iterator

C++ defines the following four type synonyms for `std::regex_token_iterator`.

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

`std::regex_token_iterator` enables us, by using indexes, to specify which capture groups we are interested in explicitly. If we don't specify the index, we will get all of the capture groups, but we can also request a specific capture group using its respective index. The -1 index is special: We can use -1 to address the text between the matches:

address the text between the matches.

```
#include <regex>

#include <iostream>
#include <string>
#include <vector>

int main(){

  std::cout << std::endl;

  // a few books
  std::string text{"Pete Becker, The C++ Standard Library Extensions, 2006:Nicolai Josuttis,

  // regular expression for a book
  std::regex regBook(R"((\w+)\s(\w+), ([\w\s\+]*), (\d{4}))");

  // get all books from text
  std::sregex_token_iterator bookItBegin(text.begin(), text.end(), regBook);
  const std::sregex_token_iterator bookItEnd;

  std::cout << "##### std::match_results ######" << "\n\n";
  while ( bookItBegin != bookItEnd){
    std::cout << *bookItBegin++ << std::endl;
  }

  std::cout << "\n\n" << "##### last name,  date of publication ######" << "\n\n";

  // get all last name and date of publication for the entries
  std::sregex_token_iterator bookItNameIssueBegin(text.begin(), text.end(), regBook, {{2, 4}}
  const std::sregex_token_iterator bookItNameIssueEnd;
  while ( bookItNameIssueBegin != bookItNameIssueEnd){
      std::cout << *bookItNameIssueBegin++ << ", ";
      std::cout << *bookItNameIssueBegin++ << std::endl;
  }

  // regular expression for a book, using negativ search
  std::regex regBookNeg(":");

  std::cout << "\n\n" << "##### get each entry, using negativ search  ######" << "\n\n";

  // get all entries, only using ":" as regular expression
  std::sregex_token_iterator bookItNegBegin(text.begin(), text.end(), regBookNeg, -1);
  const std::sregex_token_iterator bookItNegEnd;
   while ( bookItNegBegin != bookItNegEnd){
       std::cout << *bookItNegBegin++ << std::endl;
   }


  std::cout << std::endl;

}
```

`bookItBegin` uses no indices and `bookItNegbegin` uses the negative index,

through both return the total capture group. `bookNameIssueBegin` only returns

the second and fourth capture group `{{2,4}}`.

This concludes our discussion of regular expressions. In the next chapter, we'll discuss input and output features of C++.