String Splitting and Tokenization

`regex_token_iterator<>` is the Ginsu knife of the text manipulation world. It slices! It dices! This section describes how to use the highly-configurable `regex_token_iterator<>` to chop up input sequences.

Overview

You initialize a `regex_token_iterator<>` with an input sequence, a regex, and some optional configuration parameters. The `regex_token_iterator<>` will use `regex_search()` to find the first place in the sequence that the regex matches. When dereferenced, the `regex_token_iterator<>` returns a *token* in the form of a `std::basic_string<>`. Which string it returns depends on the configuration parameters. By default it returns a string corresponding to the full match, but it could also return a string corresponding to a particular marked sub-expression, or even the part of the sequence that *didn't* match. When you increment the `regex_token_iterator<>`, it will move to the next token. Which token is next depends on the configuration parameters. It could simply be a different marked sub-expression in the current match, or it could be part or all of the next match. Or it could be the part that *didn't* match.

As you can see, `regex_token_iterator<>` can do a lot. That makes it hard to describe, but some examples should make it clear.

Example 1: Simple Tokenization

This example uses `regex_token_iterator<>` to chop a sequence into a series of tokens consisting of words.

```
std::string input("This is his face");
sregex re = +_w;                      // find a word

// iterate over all the words in the input
sregex_token_iterator begin( input.begin(), input.end(), re ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
This
is
his
face
```

Example 2: Simple Tokenization, Reloaded

This example also uses `regex_token_iterator<>` to chop a sequence into a series of tokens consisting of words, but it uses the regex as a delimiter. When we pass a `-1` as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens those parts of the input that *didn't* match the regex.

```
std::string input("This is his face");
sregex re = +_s;                      // find white space

// iterate over all non-white space in the input. Note the -1 below:
sregex_token_iterator begin( input.begin(), input.end(), re, -1 ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
This
is
his
face
```

Example 3: Simple Tokenization, Revolutions

This example also uses `regex_token_iterator<>` to chop a sequence containing a bunch of dates into a series of tokens consisting of just the years. When we pass a positive integer *`N`* as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens only the *`N`*-th marked sub-expression of each match.

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // find a date

// iterate over all the years in the input. Note the 3 below, corresponding to the 3rd sub-expressior
sregex_token_iterator begin( input.begin(), input.end(), re, 3 ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
2003
1999
1981
```

Example 4: Not-So-Simple Tokenization

This example is like the previous one, except that instead of tokenizing just the years, this program turns the days, months and years into tokens. When we pass an array of integers *`{I,J,...}`* as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens the *`I`*-th, *`J`*-th, etc. marked sub-expression of each match.

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // find a date

// iterate over the days, months and years in the input
int const sub_matches[] = { 2, 1, 3 }; // day, month, year
sregex_token_iterator begin( input.begin(), input.end(), re, sub_matches ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
02
01
2003
23
04
1999
13
11
1981
```

The `sub_matches` array instructs the `regex_token_iterator<>` to first take the value of the 2nd sub-match, then the 1st sub-match, and finally the 3rd. Incrementing the iterator again instructs it to use `regex_search()` again to find the next match. At that point, the process repeats -- the token iterator takes the value of the 2nd sub-match, then the 1st, et cetera.