Compare Ranges

The functions described below allow us to check the degree of equality between ranges.

With `std::equal`, we can compare ranges. `std::lexicographical_compare` and `std::mismatch` compute which range is the smallest one.

`equal`: checks if both ranges are equal.

```
bool equal(InpIt first1, InpIt last1, InpIt first2)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2)

bool equal(InpIt first1, InpIt last1, InpIt first2, BiPre pred)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, BiPre pred)

bool equal(InpIt first1, InpIt last1, InpIt first2, InpIt last2)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, FwdIt last2)

bool equal(InpIt first1, InpIt last1, InpIt first2, InpIt last2, BiPre pred)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, FwdIt last2, BiPre pred)
```

`lexicographical_compare`: checks if the first range is smaller than the second.

```
bool lexicographical_compare(InpIt first1, InpIt last1, InpIt first2, InpIt last2)
bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, FwdIt last2

bool lexicographical_compare(InpIt first1, InpIt last1, InpIt first2, InpIt last2, BiPre pred
bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, FwdIt last2
```

`mismatch`: finds the first position at which both ranges are not equal.

```
pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1, InpIt first2)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1, InpIt first2, BiPre pred)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last2, FwdIt first2, BiPre pred)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1, InpIt first2, InpIt last2)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, FwdIt last2)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1, InpIt first2, InpIt last2, BiPre pred)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, FwdIt last2,
```

The algorithms take input iterators and eventually a binary predicate. `std::mismatch` returns a pair `pa` of input iterators as its result. `pa.first` holds an input iterator for the first element that is not equal. `pa.second` holds the corresponding input iterator for the second range. If both ranges are identical, we get two end iterators.

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>

int main(){

  std::cout << std::boolalpha << std::endl;

  std::string str1{"Only For Testing Purpose."};
  std::string str2{"only for testing purpose."};

  std::cout << "str1: " << str1 << std::endl;
  std::cout << "str2: " << str2 << std::endl;

  std::cout << std::endl;

  std::cout << "std::equal(str1.begin(), str1.end(), str2.begin()): " << std::equal(str1.begi
  std::cout << "std::equal(str1.begin(), str1.end(), str2.begin(), [](char c1, char c2){ retu
            << std::equal(str1.begin(), str1.end(), str2.begin(), [](char c1, char c2){ retu

  std::cout << std::endl;

  str1= {"Only for testing Purpose."};
  str2= {"Only for testing purpose."};

  std::cout << "str1: " << str1 << std::endl;
  std::cout << "str2: " << str2 << std::endl;

  std::cout << std::endl;

  auto pair= std::mismatch(str1.begin(), str1.end(), str2.begin());
  if ( pair.first == str1.end() ){
    std::cout << "str1 and str2 are equal" << std::endl;
  }
  else{
    std::cout << "str1 and str2 are different at position " << std::distance(str1.begin(), pa
              << " with (" << *pair.first << ", " << *pair.second << ")" <<  std::endl;
  }

  auto pair2= std::mismatch(str1.begin(), str1.end(), str2.begin(), [](char c1, char c2){ ret
  if ( pair2.first == str1.end() ){
    std::cout << "str1 and str2 are equal" << std::endl;
  }
  else{
    std::cout << "str1 and str2 are different at position " << std::distance(str1.begin(), pa
              << " with(" << *pair2.first << ", " << *pair2.second << ")" <<  std::endl;
  }
  std::cout << std::endl;


}
```

std::equal, std::lexicographical_compare, and std::mismatch

In the next lesson, we’ll use algorithms that are useful when we need to acquire a sub-range from an existing range.