【C++】Day37(2)

Class	C++
Date	@January 12, 2022
Material	
Series Number	
Summary	

【Ch10】Generic Algorirthms

10.1 Overview

Most of the algorithms are defined in the algorithm header. The library also defines a set of generic numeric algorithms that are defined in the numeric header.

In general, the algorithms do not work directly on a container. Instead, they operate by traversing a range of elements bounded by two iterators. Typically, as the algorithm traversese the range, it does something with each element.

How the Algorithms Work

To see how the algorithms can be used on varying types of containers, let's look a bit more closely at `find`. Its job is to find a particular element in an unsorted sequence of elements. Conceptually, we can list the steps `find` must take:

1. It accesses the first element in the sequence
2. It compares the element to the value we want
3. If this element matches the one we want, `find` returns a value that identifies this element
4. Otherwise, `find` advances to the next element and repeats steps 2 and 3
5. `find` must stop when it has reached the end of the sequence
6. If find gets to the end of the sequence, it needs to return a value indicating that the element was not found. This value and the one returned from step 3 must have compatible types.

None of these operations depends on the type of the container that holds the elements.

Exercise

Exercise 10.1: The `algorithm` header defines a function named `count` that, like `find`, takes a pair of iterators and a value. `count` returns a count of how often that value appears. Read a sequence of `int`s into a `vector` and print the `count` of how many elements have a given value.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char **argv) {
  std::vector<int> int_vec;
  int buff;
  while(std::cin >> buff)
    int_vec.push_back(buff);

  std::cout << "The vector contains " << count(int_vec.begin(), int_vec.end(), 3) << " numer 3.";
  return 0;
}
```

Exercise 10.2: Repeat the previous program, but read values into a `list` of `string`s.

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

int main(int argc, char **argv) {
  std::list<std::string> string_list;
  std::string buff;
  while(std::cin >> buff)
    string_list.push_back(buff);

  std::cout << "The list contains " << count(string_list.cbegin(), string_list.cend(), "Arthur") << " name Arthur.";
  return 0;
}
```

10.2 A First Look at the Algorithms

With only a few exceptions, the algorithms operate over a range of elements. We'll refer to this range as the "input range." The algorithms that take an input range always use their first two parameters to denote that range. These parameters are iterators denoting the first and one past the last elements to process.

10.2.1 Read-Only Algorithms

A number of alrogithms read, but never write to, the elements in their input range.

The `find` function is one such algorithm, as is the `count` function we used in the previous exercise.

Another read-only algorithm is `accumulate`, which is defined in the `numeric` header. The accumulate function takes three arguments. The first two specify a range of elements to sum. The third is an initial value for the sum.

Assuming `vec` is a sequence of integers, the following

```
//sum the elements in vec starting the summatyion with the initial value 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

sets `sum` equal to the sum of the elements in `vec`, using 0 as the starting point for the summation.

Note: The type of the thrid argument to accumulate determine which addition operator is used and is the type that accumulate returns.

Algorithms and Element Types

The fact that accumulate uses its third argument as the starting point for the summation has an important implication: It must be possible to add the element type to the type of the sum. That is, the elements in the sequence must match or be convertible to the type of the third argument.

In this example, the elements in `vec` might be `int`, or they might be `double`, or `long long`, or any other type that can be added to an `int`.

As another example, because string has a `+` operator, we can concatenate the elements of a vector of strings by calling `accumulate`:

```
string sum = accumulate(str.cbegin(), str.cend(), string(""));
```

Note that we explicitly create a string as the third parameter. Passing the empty string as a string literal would be a compile-time error:

```
//error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
```

Best Practice

Ordinarily it is best to use `cbegin()` and `cend()` with algorithms that read, but do not write, the elements. However, if you plan to use the iterator returned by the algorithm to change an element's value, then you need to pass `begin()` and `end()`.

Algorithms that Operate on Two Sequences

Another read-only algorithm is `equal`, which lets us determine whether two sequences hold the same values. It compares each element from the first sequence to the corresponding element in the second.

It returns true if the corresponding elements are equal, false otherwise. The algorithm takes three iterators: The first two denote the range of elements in the first sequence; the third denotes the first element in the

second sequence.

```
//roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

`equal` makes one critically important assumption: It assumes that the second sequence is at least as big as the first. It assumes that there is a corresponding element for each of those elements in the second sequence.

Warning: Algorithms that take a single iterator denoting a second sequence assume that the second sequence is at least as large as the first.

Exercise

Exercise 10.3: Use `accumulate` to sum the elements in a `vector<int>`.

```
#include <iostream>
#include <vector>
#include <string>
#include <numeric>

int main(int argc, char **argv) {
  std::vector<int> int_vec;
  int temp;
  while(std::cin >> temp)
    int_vec.push_back(temp);
  std::cout << accumulate(int_vec.cbegin(), int_vec.cend(), 0) << std::endl;
  return 0;
}
```

Exercise 10.4: Assuming `v` is a `vector<double>`, what, if anything, is wrong with calling `accumulate(v.cbegin(), v.cend(), 0)`?

The third parameter determines which `+` operator is used and type is being added. In thsi case, the type of the literal `0` is an int, thus all of the elements will be converted into `int` and then added.

Exercise 10.5: In the call to `equal` on rosters, what would happen if both rosters held C-style strings, rather than library `string`s?

```
#include <vector>
#include <list>
#include <string>
#include <iostream>

int main() {
  std::vector<const char *> roster1{"abc", "def", "ghi"};
  std::list<const char *> roster2{"abc", "def", "ghi"};
  // Two C-style strings, compare the pointer (addresses of the two strings)
  std::cout << "The two sequences are "
            << (std::equal(roster1.cbegin(), roster1.cend(), roster2.cbegin()) ?
```

```
                "equal." : "not equal.") << std::endl;  // not equal

  // Two library strings, compare the value
  std::vector<std::string> roster3{"abc", "def", "ghi"};
  std::list<std::string> roster4{"abc", "def", "ghi"};
  std::cout << "The two sequences are "
            << (std::equal(roster3.cbegin(), roster3.cend(), roster4.cbegin()) ?
                "equal." : "not equal.") << std::endl;  // equal

  // One library string and one C-style string, compare the value
  std::cout << "The two sequences are "
            << (std::equal(roster1.cbegin(), roster1.cend(), roster4.cbegin()) ?
                "equal." : "not equal.") << std::endl;  // equal

  return 0;
}
```