【C++】Day39

Class	C++
Date	@January 17, 2022
Material	
Series Number	
Summary	

【Ch10】Generic Algorithms

10.3 Customizing Operations

10.3.1 Passing a Function to an Algorithm

Assume that we want to print the vector after we call `elimDups`.

We want to see the words ordered by their size, and then alphabetically within each size. To reorder the vector by length, we'll use a second, overloaded version of `sort`. This version of sort takes a third argument that is a predicate.

Predicates

A predicate is an expression that can be called and that returns a value that can be used as a condition. The predicates used by library algorithms are either unary predicates(have a single parameter) or binary predicates(have two parameters).

The algorithms that take predicates call the given predicate on the elements in the input range. As a result, it must be possible to convert the element type to the parameter type of the predicate.

Sorting Algorithms

When we sort by size, we also want to maintain alphabetic order among the elements that have the same length. To keep the words of the same length in alphabetical order we can use the stable_sort algorithm. A stable sort maintains the original order among equal elements.

```
elimDups(words); //put words in alphabetical order and remove duplicates
//resort the vector by the size of the strings. If two strings are of the sams size, maintain the alphabetic order.
stable_sort(words.begin(), words.end(), isShorter);
for(const auto &s : wrods)
    cout << s << " ";
cout << endl;
```

Exercise

Exercise 10.11: Write a program that uses `stable_sort` and `isShorter` to sort a `vector` passed to your version of `elimDups`. Print the `vector` to verify that your program is correct.

See 10_11.cpp for code

Exercise 10.13: The library defines an algorithm named `partition` that takes a predicate and partitions the container so that values for which the predicate is `true` appear in the first part and those for which the predicate is `false` appear in the second part. The algorithm returns an iterator just past the last element for which the predicate returned `true`. Write a function that takes a `string` and returns a `bool` indicating whether the `string` has five characters or more. Use that function to partition `words`. Print the elements that have five or more characters.

See 10_13.cpp for code

10.3.2 Lambda Expressions

Introducing Lambdas

We can pass any kind of callable object to an algorithm. An object or expression is callable if we can apply the call operator `()` to it.

That is, if `e` is a callable expression, we can write `e(args)` where `args` is a comma-separated list of zero or more arguments.

A lambda expression represents a callable unit of code. It can be though of as an unnamed, inline function.

- Like any function, a lambda has a return type, a parameter list, and a function body.
- Unlike a function, lambdas may be defined inside a function.

A lambda expression has the form

```
[capture list] (parameter list) ->return type {function body}
```

where

- Capture list is an (often empty) list of local variables defined in the enclosing function
- Return type, parameter list, and function body are the same as in any ordinary function.

However, unlike ordinary functions, a lambda must use a trailing return to specify its return type.

We can omit either or both of the parameter list and return type but must always include the capture list and function body:

```
auto f = [] { return 42; };
```

Here, we've defined `f` as a callable object that takes no arguments and returns 42.

We call a lambda the same way we call a function by using the call operator:

```
cout << f() << endl; //prints 42
```

If we omit the return type, the lambda has an inferred return type that depends on the code in the function body.

If the function body is just a return statement, the return type is inferred from the type of the expression that is returned. Otherwise, the return type is void.

Note: Lambdas with function bodies that contain anything other than a single return statement that do not specify a return type return void.

Passing Arguments to a Lambda

Unlike ordinary functions, a lambda may not have default arguments. Therefore, a call to a lambda always has as many arguments as the lambda has parameters.

We can write a lambda that behaves like our `isShorter` function:

```
[] (const string &a, const string &b) -> bool { return a.size() < b.size(); };
```

We can rewrite our call to `stable_sort` to use this lambda as follows:

```
//sort words by size, but maintain alphabetical order for words of the same size
stable_size(words.begin(), words.end(), [] (const string &a, const string &b) -> bool { return a.size() < b.size(); });
```

Using the Capture List

Although a lambda may appear inside a function, it can use variables local to that function only if it specifies which variables it intends to use.

A lambda specifies the variables it will use by including those local variables in its capture list. The capture list directs the lambda to include information needed to access those variables within the lambda itself.

In this case, our lambda will capture `sz` and will have a single string parameter. The body of our lambda will compare the given string's size with the captured value of `sz`

```
[sz] (const string &a) -> bool { return a.size() >= sz; };
```

Note: A lambda may use a variable local to its surrounding function only if the lambda captures that variable in its capture list.

Note: The capture list is used for local nonstatic variables only; lambdas can use local statics and variables declared outside the function directly.