STL Algorithms

Søren Hansen <sha@ase.au.dk>
V1.2

Introduction

In this exercise we are going to read data from a file, manipulate and write them back. The idea is to use STL "power" to handel all aspects of the tasks.

In the same directory as this document you will find the directory `algo.cpp`, which is basis for the first exercise.

Exercise 1 Basic STL

Exercise 1.1 `istream_operator` and `ostream_operator`

Exercise 1.1.1 Updating file reading

Inspect the code brought to your disposal. Currently all `Product` class objects are read in function `productDBRead()` using an old style loop.

Use the newly learned `std::istream_iterator` to read the file. (Hint: Use `std::back_inserter()` - Why?)

Remember to test and verify.

Exercise 1.1.2 Updating printout

The same old style is used for writing to `std::cout` in function `printAll()` - update the function such that it uses `std::ostream_iterator`.

Exercise 1.1.3 Adding an item

Currently we can only read from the file and print out the contents. Next thing to do would naturally be to make it possible to add an item.

Add the missing code in `addItem()`.

Exercise 1.1.4 Writing product list to file

Having performed changes to the product list, we need to be able to save it for future use.

Use `std::ostream_iterator` in `productDBWrite()` to perform the deed.

Exercise 1.2 Algorithmic manipulations

Exercise 1.2.1 To few items sold...

If you analyze the product list from a sales perspective, it becomes evident that not items sells equally well, which obviously is not a surprise. However, in this trial of times we wish only to have products in stock that actually sell well.

Complete the function `printPoorlySellingProducts()` with a printout that only contains those products that have sold fewer than 10 in all.

Use `std::remove_copy_if()` for copying those that are desired and `std::ostream_iterator` for writing to `std::cout`

Exercise 1.2.2 Discount

In the following two exercises a discount is set on the products using two different approaches.

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Exercise Manipulating each element using `std::for_each()`

In this first exercise we are going to use `std::for_each()` and for this to work we need a *functor* having the signature `void operator()(Product& p)`.

For each call it must set a new price on the product. Upon constructing your functor ensure that it takes an argument such that you via it, can designate the discount you want on each product.

Implement your solution in `addDiscountUsingForEach()`.

Exercise Transforming each element using `std::transform()`

Having tried using `std::for_each()` lets try using `std::transform()`. The idea in this exercise is to pass the (1) product list, (2) a discount calculating functor and (3) `std::ostream_iterator` for printing out directly.

The functor should have the signature `Product operator()(Product p)`. As with the previous exercise, a constructor is needed that takes the discount in question as input.

Note that this exercise differs from the previous in that the transformed products are printed to a list and thus the changes placed on them are not permanent.

Implement your solution in function `addDiscountUsingTransform()`.

Exercise 1.2.3 Calculating the total amount of products sold

An interesting metric could be the total amount of products sold. To calculate this we will be using `std::transform()`, `std::back_inserter()` and `std::accumulate()`.

To help out we will use an intermediate `std::vector<unsigned int>` that can hold the *sold product count* for each product.

The idea is to populate the `std::vector<unsigned int>` using `std::transform()` (& `std::back_inserter`), but since its a specific method `sold()` that must be called on each `Product` we also need to use `std::mem_fun_ref()`.

Finally use `std::accumulate` on the `std::vector<int>` to calculate the total count value.

Exercise 1.2.4 Setting discount using lambda

Add a method `void setDiscount(float discount)` to your class `Product`.

Add a new function in which you use an appropriate algorithm to iterate through the product list and call the aforementioned `setDiscount()` method with a value of your choice.

Remember to add a 9'th menu point from where you call your new function.

Exercise 1.3 Reflection

Discuss the pros and cons of using algorithms. E.g. In your opinion does the use improve clarity or does it add to the clutter?

Include code snippets to substantiate your stance.

Exercise 2 Creating my own Iterator

Revisit the template class `MyArray<>` you created in the *Template* lab challenge. Extend it such that you may use it with *STL Algorithms*. This means implementing an iterator. To prove it works verify it with a couple of STL algorithms. For instance `std::copy()` and `std::ostream_iterator()`.

Hints: Have a look at `http://www.cplusplus.com/reference/iterator/iterator/`.

One of the important aspects of implementing and especially using an iterator stems from its category. Which category of iterators is this particular iterator member of and why is this the case?

Exercise 3 Creating my own iterator adapter (OPTIONAL)

In this exercise you will be trying to create an iterator adapter, such that you create a construct that can be used in place of what you commonly would use iterators for. E.g. like the iterator from a `std::vector`.

In the second exercise you will create an implementation that from an output stand point produces the same result but albeit differently.

Exercise 3.1 The challenging approach

In STL there are several different iterator adaptors, in this exercise we are going to create one similar to the `back_inserter()`. The difference being that instead of calling `push_back()` on the container it has been passed, it now has to be a callable entity that takes precisely one parameter being what the iterator is passed. The iterator in question is thus an *output iterator*.

The amount of code needed to implement this exercise is fairly inconsequential, but the challenge is to know exactly what those few lines of code need to be.

To show an example of what must work:

Listing 3.1: Using the iterator adapter

```cpp
/* Implementation of Wrapper<> */
/* Should be here...*/

class Test
{
public:
  void printExt(int i) const
  {
    std::cout << "i: " << i << std::endl;
  }
};

Test t;
Wrapper<int>  w(boost::bind(&Test::printExt,  t, _1));
std::vector<int> v({ 1,3,4,6,8}); // C++11 Initialization of vector

// Copying each and every element from vector to our output iterator
    adaptor Wrapper<>
std::copy(v.begin(), v.end(), w);
```

Note that `boost::bind()` or `std::bind` may not be known and thus something you will have to read about it... A hint, the input parameter to the `Wrapper<>(...)` constructor is `std::function`.

Exercise 3.2 The easy approach

Now do the same using a lambda...

Listing 3.2: Using lambda for calling specific function

```cpp
    class Test
    {
      public:
      void printExt(int i) const
      {
        std::cout << "i: " << i << std::endl; /* Kept overly simple for
            pedagogical reasons */
      }
    };

    Test t;

    std::for_each(v.begin(), v.end(), /* Some lambda expression */ );
```

Exercise 3.3 Reflection

Questions to reflect upon:

- Exactly in what way does these two implementation differ and thus their usage?

- Could the second approach have been solved using a functor? If so what would it look like?
- Discuss whether the `class Wrapper<>` has its benefits or not as opposed to, lets say, the lambda approach.