Assignment #4

CS 225, Summer 2018

Due Date:	21st July 2018 2359hrs
Topics covered:	Class template, function template, template specialization, Meta-programming
Deliverables:	To submit one zip file containing the one file `functional-list.h`
Compiler flags :	For gnu, it would be `-std=c++1z -Wall -Wextra -Werror -pedantic` For Visual Studio 2017 (Command line), it would be `/W4 /WX /nologo /EHa`
Objectives:	To practise the various techniques when implementing templates.

1 Programming Statement: To provide facilities for a compile-time list of non-type template arguments

We begin by consider the following incomplete declaration of a class template within a class template.

```
template<typename T, typename Comp=std::less<T> >
struct Facility
{
    template<T ... list>
    struct List
    {
        ...
    };
};
```

Given the structure declare above, we are able to instantiated lists of integral types. The following shows some of the examples.

```
template<int ... intlist>
using IntList = typename Facility<int>::List<intlist...>;

using List1 = IntList<1,2,3>; /* List of 1,2,3*/
using List2 =  IntList<>; /* Empty List*/
```

The student is then expected to implement the following facilities (or rather, meta-functions) to manipulate such compile-time lists.

1. `print`
2. `DotProduct`
3. `Concat`

4. `Min`
5. `RemoveFirst`
6. `Sort`
7. `Partition`
8. `QuickSort`

The following sections will explain the use of each of the above in turn.

1.1 `print`

This is really a static member function of the function template `List` that prints out each element of the parameter pack in order. Examples are given below:

```
template<short ... list>
using ShortList = typename Facility<short>::List<list...>;

using List1 = ShortList<4,9,6>; /* List of 4,9,6*/
using List2 =  ShortList<>; /* Empty List*/

List1::print(); /* prints "4 9 6", including the double quotes*/
List2::print(); /* prints "Empty List", including the double quotes*/
```

A special case should be made when `T` is char. See the example below:

```
template<char ... list>
using CharList = typename Facility<char>::List<list...>;

using List1 = ShortList<'g','g','w','p'>; /* List of 4 chars*/
using List2 =  ShortList<>; /* Empty List*/

List1::print(); /* prints "ggwp", including the double quotes*/
List2::print(); /* prints "Empty List", including the double quotes*/
```

The student is expected to be able to support printing of lists of up to 500 elements. However, an additional bonus of 10% is given if your solution can print up to 20,000 elements.

1.2 `DotProduct`

The name suggests itself. `DotProduct` is a metafunction that in the end computes the dot product of two lists of the same type. The following shows the example of how it is used:

```
template<short ... list>
using ShortList = typename Facility<short>::List<list...>;
using ShortFacility = Facility<short>;
using FirstVec = ShortList<2, 8, 10>;
using SecondVec = ShortList<-20, -5, 20>;
std::cout << ShortFacility::DotProduct<FirstVec, SecondVec>::result << std::↩
    endl; /* prints 120*/
```

Attempts to perform DotProduct on Lists of unequal number of items or different types should result in compile error.

1.3 Concat

`Concat` is a metafunction that takes in two lists and return a list that's the concatenation of the two. The following shows how `Concat` works:

```
template<int ... list>
using IntList = typename Facility<int>::List<list...>;
using IntFacility = Facility<int>;
using FirstList = IntList<2, 8, 10>;
using SecondList = IntList<-20, -5, 20>;
IntFacility::Concat<FirstVec, SecondVec>::result::print();
/* prints out "2 8 10 -20 -5 20" */
```

Of course, we should be able to concatenate lists of different number of items, provided that we are dealing with lists of the same type.

1.4 RemoveFirst

`RemoveFirst` is a metafunction that takes a list and an item. It removes the first occurence of the item on the list. If the item is not found, then the list remains as what it ought to be. The following shows the use of `RemoveFirst`.

```
template<int ... list>
using IntList = typename Facility<int>::List<list...>;
using IntFacility = Facility<int>;
using List = IntList<2, 8, 2, 3, 5, 10, 8, 5>;
IntFacility::RemoveFirst<2, List>::result::print();
/* prints "8 2 3 5 10 8 5" */
IntFacility::RemoveFirst<8, List>::result::print();
/* prints "2 2 3 5 10 8 5"*/
IntFacility::RemoveFirst<5, List>::result::print();
/* prints "2 8 2 3 10 8 5" */
IntFacility::RemoveFirst<9, List>::result::print();
/* prints "2 8 2 3 5 10 8 5" */
```

1.5 Min

`Min` is a metafunction that takes a list and returns the smallest item on the list. The specific definition of "smaller" depends on the type of `Comp`,which is defaulted to `std::less<T>`.

```
template<int ... list>
using IntList = typename Facility<int>::List<list...>;
using IntFacility = Facility<int>;
using List1 = IntList<2, 8, 2, 3>;
using EmptyList = IntList<>;
std::cout << IntFacility::Min<List1>::result << std::endl;
/* prints 2 */
/* std::cout << Min<> << std::endl;  - compile error */
```

1.6 Sort

`Sort` is a metafunction that takes a list and returns the sorted ascending order of the list.

Again, the specific definition of "smaller" depends on the type of `Comp`.

The student should implement selection sort algorithm in `Sort`. The algorithm of selection sort is simple. It can be implemented using a combination of `Min` and `Concat`.

```
1
2 \begin{lstlisting}
3 template<int ... list>
4 using IntList = typename Facility<int>::List<list...>;
5 using IntFacility = Facility<int>;
6 using List1 = IntList<2, 8, 2, 3>;
7 IntFacility::Sort<List>::result::print(); /* prints "2 2 3 8"*/
```

1.7 Partition

`Partition` is a metafunction that splits a list into 2 halves using the first element as a pivot. For example, if the list is currently 5, 4, 8, 9, 2. The first element is 5. So the list will be split into 4,2 and 8,9 respectively. The first half consists of items strictly smaller than the pivot while the second half consists of items strictly larger or equal to the pivot. The following shows the use of the Partition algorithm.

```
1 template<int ... list>
2 using IntList = typename Facility<int>::List<list...>;
3 using IntFacility = Facility<int>;
4 using EmptyList = IntList<>;
5 using Singleton = IntList<1>;
6 using List1 = IntList<2, 8, 1, 3>;
7 using List2 = IntList<1, 3, 5>;
8 IntFacility::Partition<List1>::FirstHalf::print(); /*prints out "1"*/
9 IntFacility::Partition<List1>::SecondHalf::print(); /*prints out "8 3"*/
10 IntFacility::Partition<List2>::FirstHalf::print();
11 /*prints out "Empty List"*/
12 IntFacility::Partition<List2>::SecondHalf::print(); /*prints out "3 5"*/
13 /* Partition<EmptyList>::FirstHalf - should be compile error. */
```

1.8 QuickSort

`QuickSort` is a metafunction that sorts a list in an ascending order based on the `Comp` type. However, it differs from `Sort` in terms of the algorithm it employs. `QuickSort` is only allowed to use `Partition` to implement it's sorting capabilities. A pseudo-code for QuickSort is given below:

```
QuickSort(list)
    if list is empty, return
    Partition list into two halves around pivot value
    list1 <- QuickSort(FirstHalve)
    list2 <- QuickSort(SecondHalve)
    return the list made up of list1 + pivot + list2
```

```
1 template<int ... list>
```

```
using IntList = typename Facility<int>::List<list...>;
using IntFacility = Facility<int>;
using List1 = IntList<2, 8, 2, 3>;
IntFacility::QuickSort<List>::result::print(); /* prints "2 2 3 8"*/
```

1.9 Rubrics

The student is expected to implement all the functionality delineated above in the file `functional-list.h`. The following shows the breakdown of each functionality:

- `print` 10%
- `DotProduct` 10%
- `Concat` 10%
- `Min` 10%
- `RemoveFirst` 15%
- `Sort` 15%
- `Partition` 15%
- `QuickSort` 15%

If the student is able to match the output, using the given driver, the student should be able to obtain 80% of the above score. Hidden tests will be done which may cost students up to 20% of the entire assignment.

The rest of the rubrics will be scored according to the usual coding guidelines that's shown in Moodle.