Workshop 5

Functions and Error Handling

In this workshop, you code a function object, a lambda expression, and exception handling.

LEARNING OUTCOMES

Upon successful completion of this workshop, you will have demonstrated the abilities to

- design and code a templated class that allocates and deallocates dynamic memory
- design and code a function object
- design and code a lambda expression
- code a member function that receives the address of another function to execute
- throw exceptions of different types
- distinguish exception types

SUBMISSION POLICY

The *in-lab* section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. If you do not attend the workshop, you can submit the *in-lab* section along with your *at-home* section (see penalties below). The *at-home* portion of the lab is due on the day that is four days after your scheduled in-lab workshop (23:59:59) (even if that day is a holiday).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.

You are responsible to back up your work regularly.

Late Submission Penalties:

- *In-lab* portion submitted late, with *at-home* portion: **0** for *in-lab*. Maximum of 7/10 for the entire workshop.
- If any of *in-lab*, *at-home* or *reflection* portions is missing, the mark for the workshop will be **0**/10.

SPECIFICATIONS – IN LAB

The solution to the in-lab part of this workshop consists of four modules:

- **w5** (supplied)
- **KVList**
- **KVPair**
- **Taxable**

Enclose all your source code within the **sict namespace** and include the necessary guards in each header file. The output from your executable running Visual Studio with the following command line argument should look like

```
Command Line : C:\Users\...\Debug\in_lab.exe PriceList.txt

Price List with G+S Taxes Included
==================================
Description:      Price Price+Tax
Soap        :       1.29      1.46
Detergent   :       3.49      3.94
Lysol       :       3.59      4.06
Kleenex     :       0.50      0.56
```

The input for testing your solution is stored in a user-prepared file. The name of the file is specified on the command line as shown in red above. The file is supplied with this workshop. The contents of this file are

```
Soap 1.29
Detergent 3.49
Lysol 3.59
Kleenex 0.50
```

Each record in the file consists of key and value fields. For the `PriceList.txt` file, the name of the product is the key and the price of the product is its value.

KVPair Module

Design and code a templated class named **KVPair** for managing a single key-value pair. Your class design includes the following public member functions:

- A default constructor that places the object in a safe empty state.
- **KVPair(const K&, const V&)** – a two-argument constructor that receives a reference to an unmodifiable key and a reference to an unmodifiable value. This function stores the received key and value in the object's instance variables.

- **template<typename F>void display(std::ostream& os, F f) const** – a templated query that inserts into the output stream **os** the current object's key (left-justified), a single space, a colon, a single space, the object's value and the object's value converted through the function **f** (both right justified) as shown above. If the current object is in a safe empty state, this query does nothing. The field width for the object's values is externally defined.

KVList Module

Design and code a templated class named **KVList** for managing a dynamically allocated list of **T** objects. Your class design includes the following public member functions:

- A default constructor that places the object in a safe empty state.
- **KVList(int n)** – a single-argument constructor that receives the number of objects in the list and allocates memory for those objects. If the number received is not positive-valued your constructor places the object in a safe empty state.
- **KVList(KVList&&)** – a move constructor.
- **~KVList()** – a destructor.
- **const T& operator[](size_t i) const** – a subscripting operator that returns an unmodifiable reference to the **i**-th element in the list. For the in-lab part of this workshop, you may assume that the index is not out of bounds.
- **template<typename F>void display(std::ostream& os, F f) const** – a templated query that inserts into the output stream **os** on a separate line each object in the list as shown above. If there are no objects in the list or the object is in a safe empty state, this query does nothing.
- **push_back(const T& t)** – a modifier that receives an unmodifiable reference to a **T** object **t** and if there is room in the list, adds the object **t** to the list. If the list is full, this function does nothing.

Your design disables copy and move assignment operations and copy construction of the list.

Taxable Module

Design and code a functor (function object) named **Taxable** that receives a value and returns the sum of the value with the tax on it. Your class design includes the following public member functions:

- **Taxable(float)** A one-argument constructor that receives the prescribed tax rate and stores it in an unmodifiable instance variable.

- **float operator()(float)** A function call operator that receives a price and returns the sum of the price and the tax added to it.

In-Lab Submission (30%)

To test and demonstrate execution of your program use the same data as shown in the output example above.

Upload your source code to your `matrix` account. Compile and run your code using the latest version of the gcc compiler and make sure that everything works properly.

Then, run the following command from your account: (replace `profname.proflastname` with your professor's Seneca userid)

~profname.proflastname/submit 345XXX_w5_lab<ENTER>

and follow the instructions. Replace **XXX** with the section letter(s) specified by your instructor.

SPECIFICATIONS – AT HOME

The at-home part of this workshop updates the following modules:

- **w5** (from in-lab)
- **KVList** (from in-lab)
- **KVPair** (from in-lab)

The output from your executable running Visual Studio with the following command line argument should look like

```
Command Line : C:\Users\...\Debug\at_home.exe PriceList.txt Student.txt

Price List with G+S Taxes Included
==================================
Description:      Price Price+Tax
Soap       :       1.29      1.46
Detergent  :       3.49      3.94
Lysol      :       3.59      4.06
Kleenex    :       0.50      0.56

Student List Letter Grades Included
==================================
Student No :      Grade    Letter
203342102  :      67.50        C+
923416789  :      83.40        A
```

```
310654789  :      56.70        D+
201352234  :      63.40        C
```

The input for testing your solution is stored in two user-prepared files. The names of the files are specified on the command line as shown in red above. Two files are supplied with this workshop.

KVList Module

Add exception reporting to this module to cover the following cases:

- The number of elements received in the constructor is not positive-valued
- The index passed to the subscripting operator is out-of-bounds

You will need to add code to handle these exceptions in the calling function (w5).

w5 Module

Add code to process a list of student grades and convert their values to letter grades. Note that the letter grades without a plus consist of the letter followed by a single space. In other words, all letter grades consist of two characters, excluding the null byte.

The code for this grade list is similar to that for the price list. The key is an **int** and the grade is a **float**. Use a lambda expression instead of a functor for your conversion function.

Introduce exception handling into your module wherever it is appropriate. Your exceptions should include command line errors as well as errors generated by the **createList()** function.

Call the **exit()** function where appropriate, using a different exit value for each type of error.

Reflection

Study your final solution, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. This should take no less than 30 minutes of your time. Explain in your own words what you have learned in completing this workshop. Include in your explanation but do not limit it to the following points (40%):

- The difference between the implementations of a functor and a lambda expression.
- A list of the exceptions that you have reported and the cases where you terminated execution normally.

To avoid deductions, refer to code in your solution as examples to support your explanations.

Include all corrections to the Quiz(zes) you have received (30%).

At-Home Submission (70%)

To test and demonstrate execution of your program use the same data as shown in the output example above.

Upload your source code to your `matrix` account. Compile and run your code using the latest version of the gcc compiler and make sure that everything works properly.

Then, run the following command from your account: (replace `profname.proflastname` with your professor's Seneca userid)

~profname.proflastname/submit 345XXX_w5_home<ENTER>

and follow the instructions. Replace **XXX** with the section letter(s) specified by your instructor.