Workshop 3

Templates

In this workshop, you design and code a couple of class templates and test them on different instantiations.

LEARNING OUTCOMES

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

- design and code a class template
- template a class variable
- specialize a class template for a particular type
- instantiate a template class
- specialize a member function of a class template to process a particular type
- derivate a template class from another template class

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

This workshop consists of three modules:

- **w3** (supplied)
- **List**
- **LVPair**

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 Sales.txt

Ticket Sales
============
Student : 25
Adult : 13
Student : 12
Adult : 6
Student : 5
Adult : 15
```

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.

For a complete solution to the in-lab part of this workshop you need to create two source files:

- **List.h** - defines a class template for a list of elements of any data type (for example, a list of **int**s)
- **LVPair.h** – defines a class template for a label-value pair (for example, a product label stored in an **std::string** object and a quantity stored in an **int** object)

List Module

Design and code a class template named **List** for managing an array of any datatype. The template parameters in order of their specification are

a) **T** - the type of any element in the array
b) **N** - the maximum number of elements in the array and

Your template includes a data member of type **size_t** (an **unsigned int**) that holds the number of elements in the array.

Your template design includes the following member functions:

- **size_t size() const** – a query that returns the number of elements stored in the array
- **const T& operator[](size_t i) const** – an overloaded operator that receives an index and returns a reference to the unmodifiable object stored at that index of the array.
- **void operator+=(const T& t)** – an overloaded operator that receives a reference to an unmodifiable object **t** of type **T**. If space to store the additional element is available, this operator appends a copy of the referenced object **t** to the set of elements currently stored in the array. If space to store the additional element is not available, this operator does nothing.

LVPair Module

Design and code a class template named **LVPair** for managing a label-value pair. The template parameters identify the types of the label and value objects that constitute an **LVPair** object:

a) **L** – the type of the label
b) **V** – the type of the value

Your template design includes the following member functions:

- **LVPair()** – default constructor – leaves the object in a safe empty state
- **LVPair(const L& label, const V& value)** – an overloaded constructor that copies the values received in its parameters into the instance variables.
- **void display(std::ostream& os) const** – a query that inserts into **os** the label and value stored in the current object separated by a space-colon-space string (**" : "**) as shown in the example above
- **std::ostream& operator<<(std::ostream& os, const LVPair& pair)** – a non-friend helper function that inserts into the **os** object the **LVPair** object referenced in the 2nd function parameter

Execution

A text file named **Sales.txt** is included in the directory containing the Visual Studio project file. When executing your solution specify this file name as a command line argument.

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_w3_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 upgrades your in-lab solution to include

a) alignment of the label and value output in pretty columnar format
b) accumulation of the values in a **List** for a specified label

To implement each upgrade, you will derive a templated class from your original templated class (one derived class from **List** and one derived class from **LVPair**) and specialize the class derived from **LVPair** as described below.

Copy your completed in-lab header files (**List.h** and **LVPair.h**) to your at-home project directory. Add your template upgrades to these header files.

The output from your solution will look something like:

```
Command Line : C:\Users\...\Debug\at_home.exe References.txt Sales.txt

Individual Index Entries
========================
Types      : 23
Pointers   : 26
References : 26
Pointers   : 150
Pointers   : 162
References : 65

Collated Index Entries
======================
Pointers   26 150 162
References 26 65

Detail Ticket Sales
===================
Student : 25
Adult   : 13
Student : 12
```

```
Adult   : 6
Student : 5
Adult   : 15

Summary of Ticket Sales
=======================
Student Tickets =   86.10
Adult Tickets   =  110.50
```

Note that the output for each file consists of two parts: a list of entries and a summary of the entries. The summary following each list of entries is the 'sum' of the entries. In the case of the `References.txt` file, this sum is a concatenation of the values for each given label. In the case of the `Sales.txt` file, the sum is the arithmetic sum of the values for each given label.

LVPair Module

Your **LVPair** module needs to include both summation and label alignment functionality.

Derive, from your original **LVPair** template class, a class template named **SummableLVPair** to manage the summation and pretty displaying of labeled values. Your derived class includes the following two *class* variables of the specified type:

- **V** – holds the initial value for starting a summation (this depends on the type of the value in the label-value pair and will be defined separately).
- **size_t** – holds the minimum field width for pretty columnar output of label-value pairs – this is the minimum number of characters needed to display anyone of the labels in a set of labels.

Enable overriding of your **LVPair::display()** member function by declaring it **virtual**. This is the only change that you need to make to your original template.

Your derived template class includes the following member functions:

- **SummableLVPair()** – default constructor – leaves the object in a safe empty state
- **SummableLVPair(const L& label, const V& v)** – an overloaded constructor that calls the base class 2-argument constructor, passes the values received to the base class and increases the stored field width if it is less than the return of characters required to display the label for all **LVPair** objects. This class assumes that the type of the first parameter has a member function named **size()**, which returns that value.
- **const V& getInitialValue()** – a class function that returns the initial value for the summation of a set of **LVPair** objects of label type **L**.
- **V sum(const L& label, const V& sum) const** – this query receives two unmodifiable references - one to a label (**label**) and another to a partially accumulated sum (**sum**) –

and returns in a **V** object the sum of the value of the current object and the partially accumulated sum.

- **void display(std::ostream& os) const** – a query that inserts into the **std::ostream** object the label and value stored in the base class. Before calling the **display()** function on the base class, this query sets the **std::ostream** object to left-justified insertion and a field width equal to that stored for objects of this class. (This field width is stored in the class variable).

Your design includes the following additional statements:

- A templated declaration that initializes the field width class variable to 0.
- A template specialization that initializes the starting value for an **LVPair<std::string, int>** type to 0.
- A template specialization that initializes the starting value for an **LVPair<std::string, std::string>** type to an empty string.
- A template specialization of the **sum()** query for **LVPair<std::string, std::string>** types that inserts a single space between concatenating strings.

List Module

Your **List** module needs to include summation functionality for a specified label.

Derive, from your original **List** template, a class template named **LVList** to manage a list of *summable* elements. The template parameters for the **LVList** template in order of their specification are

a) **T** - the type of any element in the array
b) **L** – the type of the specified label
c) **V** – the type of the summation value
d) **N** - the maximum number of elements in the array and

Your derived template adds one member function to the list hierarchy:

- **V accumulate(const L& label) const** – a query that receives a reference to an unmodifiable label (**L**) object and returns the sum of the values of all elements in the current **LVList** object that have label of the specified name (**label**) in a locally created value (**V**) object. This function initializes the accumulator (**V**) object to the initial value for objects of the label-value pair (**T**) (see below) and then accumulates the values by calling the append query on each element in the list stored in the base class. (Hint: to access the **size()** and **operator[]()** member functions of the base class, cast the current object (***this**) to a reference to the base class sub-object – **((List<T, N>&)*this)** – before calling the member function on the base class sub-object.)

Main Module

Your main module needs to identify the types that the compiler needs to apply in creating classes to process the input files. Use the information from your own template design to identify the types to be passed as arguments to the templates declarations.

Complete the main function supplied with the at-home project file by adding 4 statements:

a) Declare an object named **references** as an instance of an **LVList** derived class. In this case, each derived class object is a **SummableLVPair** object, which consists of a label of **std::string** type and a value of **std::string** type.
b) From each record received as input from the `References.txt` file create a temporary **SummableLVPair** object and add it to the **references** object.
c) Declare an object named **ticketSales** as an instance of an **LVList** derived class. In this case, each derived class object is a **SummableLVPair** object, which consists of a label of **std::string** type and a value of **int** type. (Hint: Examine the declaration of the **ticketSales** object in **in_lab/w3.cpp** for an example of the syntax for this instruction.)
d) From each record received as input from the `Sales.txt` file create a temporary **SummableLVPair** object and add it to the **ticketSales** object.

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 reason for specializing the **sum()** member function.
- The reason for specializing the initial value for a summation.
- The reason for defining the class variable outside the class definition.

To avoid deductions, refer to code in your solution as an example of your implementation of the concepts that you describe.

Include all corrections to the Quiz 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_w3_home<ENTER>
```

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