**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). **In order to get credit for the *in-lab* portion, you must be present in the lab for the entire duration of the lab.**

The *at-home* portion of the workshop 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, student number and the date of completion (use the following template):

// Name:

// Seneca Student ID:

// Seneca email:

// Date of completion:

//

// I confirm that the content of this file is created by me,

// with the exception of the parts provided to me by my professor.

You are responsible to back up your work regularly.

**Late Submission Penalties**

The workshop can be submitted up to **1 (one) day** late (the day that is 5 days after the lab period); submissions received on this day are considered **late** and are subject to penalties:

* only *in-lab* portion submitted late (after the end of the lab period): 0 for *in-lab* portion, max 7/10 for the entire workshop.
* only *at-home* portion submitted late (more than 4 days after the lab period): max 4 for the *at-home* portion, max 7/10 for the entire workshop.
* both *in-lab* **and** *at-home* portions submitted late: max 4/10 for the entire workshop.
* when the submission closes, if the workshop is not complete, the mark for the entire workshop will be 0/10. The workshop is considered complete if there are two separate submissions (*in-lab* submission and *at-home* submission) containing the *in-lab code*, *at-home code* and *reflection*.

The submission is considered closed at the end of the day that is 5 (five) days after the lab period.

***In-Lab***

This workshop consists of three modules:

* w3 (supplied)
* List
* LVPair

Enclose all your source code within the sdds namespace and include the necessary guards in each header file.

**w3 Module (supplied)**

**Do not modify this module!** Look at the code and make sure you understand it.

**List Module**

This module defines a class template for a collection of elements of any data type (for example, a list of int, or a list Student, etc.)

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

* T: the type of any element in the array
* N: the maximum number of elements in the array (an integer without sign)

Your design should be able to distinguish between

* the capacity of the array (N)
* the number of elements added to the list. Initially the list is empty.

***Public Members***

* size\_t size() const: returns the number of elements in the list
* const T& operator[](size\_t i) const: returns the element at index i (assume the parameter is valid).
* void operator+=(const T& tt): if the list didn't reach the capacity, add a copy of the parameter to the array. Otherwise, do 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:

* L: the type of the label
* V: the type of the value

***Public Members***

* LVPair(): sets the object in an empty state
* LVPair(const L& aa, const V& bb): copies the values received in the parameters into the instance variables.
* const L& key() const: returns the **key** component of the pair
* const V& value() const: returns the **value** component of the pair
* void display(std::ostream& os) const: inserts into the parameter the stored values in the following format

LABEL : VALUE<endl>

***Free Helpers***

* std::ostream& operator<<(std::ostream& os, const sdds::LVPair<L, V>& pair): calls the function LVPair<L, V>::display() to insert a pair into the stream.

**Sample Output**

When the program is started with the command (the file sales.txt is provided):

w3.exe sales.txt

the output should look like:

Command Line:

--------------------------

1: w3.exe

2: sales.txt

--------------------------

Detail Ticket Sales

--------------------------

Student : 25

Adult : 13

Student : 12

Adult : 6

Student : 5

Adult : 15

**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 g++ compiler (available at /usr/local/gcc/9.1.0/bin/g++) 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 305XXX\_w3\_lab

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

**⚠️Important:** Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

***At-Home***

The *at-home* part of this workshop upgrades your *in-lab* solution to include

* alignment of the label and value output in pretty columnar format
* accumulation of the values stored 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.

**LVPair Module**

The LVPair module must be upgraded to include both summation and label alignment functionality.

Modify the function display() in the class LVPair to enable polymorphism on it (make it virtual). This is the only change that you need to make to your original template.

Add to this module a class template named SummableLVPair to manage the summation and pretty displaying of labeled values.

This class is derived from LVPair<L, V>, and receives 2 template parameters:

* L: the type of the label
* V: the type of the value

***Static Private Members for SummableLVPair***

* an object of type V that holds the initial value for starting a summation. The initial value depends on the type of the value in the label-value pair and will be defined separately.
* a variable of type size\_t that holds the minimum field width for pretty columnar output of label-value pairs (initialize it with 0). This is the minimum number of characters needed to display any of the labels in a set of labels.

This value must be updated every time a new pair is added to the collection.

***Static Public Members for SummableLVPair***

* static const V& getInitialValue(): return the initital value stored.

***Public Members for SummableLVPair***

* default constructor
* SummableLVPair(const L& lbl, const V& val): stores the pair in the collection, and updates the field width if necessary. This functions assumes that the type L supports a function named size() that returns the number of characters required to display lbl.
* V sum(const L& lbl, const V& val) const:
  + If the label of the pair stored in the current instance is lbl, then add the value of the pair and val together and return the result. Use + for addition.
  + Otherwise, return val.
* void display(std::ostream& os) const: set the alignment to left and the field width and then call display() from the parent class. At the end, restore the alignment to right.

***Specializations***

* for SummableLVPair<std::string, std::string>, the initial value for summation should be set to empty string ("").
* for SummableLVPair<std::string, int>, the initial value for summation should be set to 0.
* for SummableLVPair<std::string, std::string>, the function sum() should concatenate the values stored using , as separator (use operator + to concatenate strings).

**List Module**

The template class List doesn't need any change.

Add to this module another **template** class called LVList, that can manage a collection of *summable* elements.

This class is derived from List<T, N>, and receives 4 template parameters:

* L: the type of a label
* V: the type summation value
* T: the type of any element in the collection
* N: the maximum number of elements in the collection (an integer without sign)

In this design, summable elements are objects of a type that supports the operation V sum(const L& l, const V& v).

***Public Members for LVList***

* V accumulate(const L& label) const: this function accumulates all the values stored in the list that have the label label into a local object of type V.
  + get the initial value from the type T and store it into a local variable of type V: this is the accumulator. In this design, the type T must have a static member called getInitialValue().
  + iterate over the collection and call the function sum() for each item (use the overloaded operator[] to access the item at index i). Store the result of sum() into the accumulator.
  + return the accumulator to the client.

Once the implementation of this module is complete, if you attempt to instantiate the class LVList using a type T that doesn't support the sum() and getInitialValue() operations, you will receive compilation errors **for that specific instantiation**.

**Sample Output**

When the program is started with the command:

w3.exe products.txt sales.txt

the output should look like:

Command Line:

--------------------------

1: w3.exe

2: products.txt

3: sales.txt

--------------------------

Individual Index Entries

--------------------------

Grocerries : tomatoes

Electronics : computer

Tools : hammer

Grocerries : lettuce

Grocerries : potatoes

Electronics : Multimedia\_Player

Electronics : HDD

Grocerries : meat

Tools : jigsaw

Collated Index Entries

--------------------------

Tools: hammer, jigsaw

Grocerries: tomatoes, lettuce, potatoes, meat

Electrnics:

Electronics: computer, Multimedia\_Player, HDD

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

**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.**

Create a **text** file named reflect.txt that contains your detailed description of the topics that you have learned in completing this particular workshop and mention any issues that caused you difficulty and how you solved them. Include in your explanation—**but do not limit it to**—the following points:

* 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.

**Quiz Reflection**

Add a section to reflect.txt called **Quiz X Reflection**. Replace the **X** with the number of the last quiz that you received and list all questions that you answered incorrectly.

Then for each incorrectly answered question write your mistake and the correct answer to that question. If you have missed the last quiz, then write all the questions and their answers.

**Submission (30% for code, 40% for reflection)**

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

Upload the source code and the reflection file to your matrix account. Compile and run your code using the latest version of the g++ compiler (available at /usr/local/gcc/9.1.0/bin/g++) 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 305XXX\_w2\_home

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

**⚠️Important:** Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.