**Workshop 3: Templates**

In this workshop, you design and code several 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 templated class variable for a particular type
* instantiate a template class
* specialize a member function of a templated class to process a particular type
* derive a templated class from another templated class

**Submission Policy**

The workshop is divided into two coding parts and one non-coding part:

* *Part 1*: worth 0% of the workshop's total mark, is optional and designed to assist you in completing the second part.
* *Part 2*: worth 100% of the workshop's total mark, is due on **Sunday at 23:59:59** of the week of your scheduled lab. Submissions of *Part 2* that do not contain the *reflection* are not considered valid submissions and are ignored.
* *reflection*: non-coding part, to be submitted together with *Part 2*. The reflection does not have marks associated to it, but can incur a **penalty of max 40% of the whole workshop's mark** if your professor deems it insufficient (you make your marks from the code, but you can lose some on the reflection).

The workshop should contain ***only work done by you this term*** or provided by your professor. Work done in another term (by you or somebody else), or work done by somebody else and not **clearly identified/cited** is considered plagiarism, in violation of the Academic Integrity Policy.

Every file that you submit must contain (as a comment) at the top **your name**, **your Seneca email**, **Seneca Student ID** and the **date** when you completed the work.

* If the file contains only your work, or work provided to you by your professor, add the following message as a comment at the top of the file:

I have done all the coding by myself and only copied the code that my professor provided to complete my workshops and assignments.

* If the file contains work that is not yours (you found it online or somebody provided it to you), **write exactly which parts of the assignment are given to you as help, who gave it to you, or which source you received it from.** By doing this you will only lose the mark for the parts you got help for, and the person helping you will be clear of any wrong doing.

**Compiling and Testing Your Program**

All your code should be compiled using this command on matrix:

/usr/local/gcc/10.2.0/bin/g++ -Wall -std=c++17 -g -o ws file1.cpp file2.cpp ...

* -Wall: compiler will report all warnings
* -std=c++17: the code will be compiled using the C++17 standard
* -g: the executable file will contain debugging symbols, allowing *valgrind* to create better reports
* -o ws: the compiled application will be named ws

After compiling and testing your code, run your program as following to check for possible memory leaks (assuming your executable name is ws):

valgrind ws

To check the output, use a program that can compare text files. Search online for such a program for your platform, or use *diff* available on matrix.

**Part 1 (0%)**

This workshop consists of five modules:

* w3 (supplied)
* Set
* Pair
* SetSummable
* PairSummable

The solution to Part 1 invokes the first three modules.

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

All the modules **you create** in this workshop must have only a header file. *🗎 Explain in the reflection why we do not split a module into \*.h and \*.cpp like you did in the previous workshops.*

In all classes that you create, you are allowed to add any **private** members that your design requires (without changing the specs)!

**w3 Module (supplied)**

**Do not modify this module!** Look at the code and make sure you understand how to instantiate a templated class.

**Set Module**

This module represents a family of collections of elements of any data type (for example, sets of ints, or sets of Students, etc.).

Design and code a class template named Set. Your template manages a statically allocated array of any datatype. The template parameters in order of their specification are:

* N: the capacity of the collection (a non-type parameter; an integer without sign). This is the maximum number of elements that can be added to the collection
* T: the type of any element in the collection

Your design keeps track of the current number of elements stored in the collection (which may differ from the capacity of the collection (N)). Initially the collection has no elements.

**This module should not use or know the type Pair!!**

***Public Members***

* size\_t size() const: returns the current number of elements in the collection
* const T& get(size\_t idx) const: returns a reference to the element at index idx of the statically allocated array (assume that the parameter is valid).
* void operator+=(const T& item): if the collection has capacity for another element, adds a copy of item to the collection. Otherwise, does nothing.

Add any other **private** members that your design requires (without changing the specs above)!

**Pair Module**

This module represents a family of *value-key* pairs.

Design and code a class template named Pair. Your template manages a single *value-key* pair. The template parameters (in order) identify the type of the value and the type of the key that constitute a Pair object:

* V: the type of the value
* K: the type of the key

**This module should not use or know the type Set!!**

***Public Members***

* default constructor
* Pair(const K& key, const V& value): copies the values referred to by the parameters into the instance variables
* const V& value() const: returns the **value** component of the pair
* const K& key() const: returns the **key** component of the pair
* void display(std::ostream& os) const: inserts into stream os the key and the value of the pair in the following format
* KEY : VALUE<endl>

***Free Helper***

* std::ostream& operator<<(std::ostream& os, const Pair<V, K>& pair): calls the member function display() on pair to insert a pair into stream os.

Add any other **private** members that your design requires (without changing the specs above)!

**Sample Output**

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

ws sales.txt

the output should look like the one from the sample\_output.txt file.

**Test Your Code**

To test the 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/10.2.0/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 345\_w3\_p1

and follow the instructions.

***This part represents a milestone in completing the workshop and is not marked!***

**Part 2 (100%)**

The second part of this workshop upgrades your Part 1 solution to

* align the key and value output in pretty columnar format
* accumulate the values stored in a Set object, for a specified key, which serves as the filter for selecting elements from the Set

To implement each upgrade, you will derive templated classes from your original templated classes (one derived class from Set and one derived class from Pair) and specialize the class derived from Pair as described below.

**Pair Module**

Modify the display() member function in the Pair module to enable inclusion polymorphism on the Pair hierarchy.

No other changes are necessary to this module.

**PairSummable Module**

A new module called PairSummable represents a Pair that has key alignment, compatibility and summation functionality.

Derive the PairSummable class template from Pair<V, K>. Your template receives 2 template parameters:

* V: the type of the value
* K: the type of the key

**This module should not use or know the type Set or SetSummable!!**

***Static Private Members for PairSummable***

* an object of type V that holds the *initial value* for a summation. The initial value varies with the type of the value in the value-key pair.
* a variable of type size\_t that holds the minimum field width for pretty columnar output of key-value pairs (initialize it with 0). This is the minimum number of characters needed to display any key in a set of keys.

This value must be updated every time a new pair is constructed.

***Public Members for PairSummable***

* default constructor
* PairSummable(const K& key, const V& value = initial): calls the base class constructor passing the two parameters to it, and updates the field width if necessary. *🗎 Explain in the reflection what = initial in the prototype means.*
  + This function assumes that the type K supports a function named size(), which returns the number of characters required to display key. Use this function to determine if the field width must be updated.
* bool isCompatibleWith(const PairSummable<V, K>& b) const: returns *true* if the parameter has the same key as the current object, *false* otherwise.
* overload the operator+= to receive a reference to an unmodifiable PairSummable object. This function adds the value of the parameter object to the value of the current object and returns a reference to the current object. Assume that the current object and the parameter have the same key.
* override the display() query to set the alignment to left and the field width to the value of the static attribute (see above, in the static members section) for all K types, then call the base class version of display(), and finally restore the alignment to right.

***Specializations***

* for V = std::string and K = std::string, the function operator+=() should concatenate the values stored using ", " as a separator (use operator + to concatenate strings), for the result shown in the sample output.

**Set Module**

The Set module doesn't require any change.

**SetSummable Module**

A new module called SetSummable represents a collection that is summable on a subset of the collection.

Derive the SetSummable class template from your Set<N, T> template. Your new template has 2 template parameters (in order):

* N: the capacity of the collection (an integer without sign)
* T: the type of any element in the collection

Your design assumes that the type T:

* has a constructor that accepts an std::string as a parameter, which specifies the filter for identifying the elements that belong to the subset.
* has a member function named isCompatibleWith, which returns *true* if the object is compatible with an object of type T; *false* otherwise.
* supports the operator+= operation, which adds an object of type T to another object of type T.

**This module should not use or know the type Pair or PairSummable!!**

***Public Member for SetSummable***

* T accumulate(const std::string& filter) const: this query accumulates into a local object of type T the subset of all the elements in the collection that satisfy filter.
  + define a local T object that satisfies filter, using the one-argument constructor. This object will serve as the accumulator.
  + iterate over the collection and add to your accumulator only those elements that are compatible with the local object.
    - check each element in the collection to determine if it is compatible with your accumulator object (use the this->get() member function to access the element)
    - if compatible, add the element to your accumulator object
  + return a copy of your accumulator object to the client.

In our **very specific instantiation**, SetSummable will manage a collection of PairSummable objects; two objects of type PairSummable are considered compatible if the have the same key. The design above is more general, and allows SetSummable to work with other types that are not PairSummable as long as they support the mentioned operations.

**Sample Output**

When the program is started with the command (the files sales.txt and products.txt are provided):

ws products.txt sales.txt

the output should look like the one from the sample\_output.txt file.

**🗎 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 and the result is suggested to be at least 150 words in length.**

Create a **text** file named reflect.txt that contains your detailed description of the topics that you have learned in completing this 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 operator+=() member function.
* why we don't need to specialize the initial value for the summation (see that the value is different when we add numbers versus when we add strings).
* the reason for defining the class variable outside the class definition.
* answers to the other questions in these specifications.

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

**Submission**

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/10.2.0/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 345\_w3\_p2

and follow the instructions.