Name: Soham Thaker SID: 011-748-159 Email: [sdthaker1@myseneca.ca](mailto:sdthaker1@myseneca.ca)

To begin with, I learned how to template a class variable and its uses. Class variables are applied to the class rather than to instances and for that reason their definitions, including initialization must be done outside of classes so that compiler can apply their value per class rather than per instance. If they were to be defined within a class, we would break the One Definition Rule which states that a definition for a function or a member must happen occur only once. Their use case includes, as we saw in this workshop where we had to accumulate the values/subset for a given key/set by looping over a collection of key-value pairs so that we can display them later in our program. Since static variables/data members exist throughout the lifetime of the program we were able to maintain their values between function calls as we accumulated the values for a given key until we reach the end of the collection of key-value pairs and then display them to the console.

I also learned how to instantiate a template. A templated class or a function has a template header that include in the angular brackets the templated type arguments which are denoted as ‘template <typename T>’ which tells that template class or template function, that it will receive at most one type that would be a generic type. However, we must be careful of the operations that, that class instance or the function might perform since if the type of T were to be a custom type, we would have to overload any operators that they use or special functions to be implemented if they perform copy/move operations. A templated class only provides programmers the ability to pass any type to a templated class and that’s about all. So, we can still perform inheritance, polymorphism, operator overloading, among other things to a templated class as usual as we can see in this workshop where we were able to inherit all the public members from the parent class and perform inclusion polymorphism among parent-child classes. Once thing to note is that if an inherited base class is templated class, we must include the template header to the derived class’s class declaration which has all the necessary number of types that the parent class requires for instantiation. Also, template arguments are a part of a template class type so every time we mention the template class be it base or derived, we must write the full type of that class like so ‘Foo<T,M> f()’ *(templated arguments are a key thing here which are to be included when instantiating an instance of templated class)* if we were to create an instance of Foo and Foo is a templated class that accepts two templated arguments. This also applies to member definitions for functions, class variables, class functions, operator overloads etc. where we must tell the compiler that we are talking about a specific function declared in that specific type *(templated class)* since templated arguments are part of the underlying type associated with that templated class. An example for a templated class’s function definition would be, ‘template <typename T, typename M> Foo<T, M>::Foo()’ which 1) specifies the template header so that the function can use those types later inside the function scope and 2) prefixes the function identifier with the full type of Foo that includes the two templated arguments along with the scope resolution operator that helps compiler decide which function is being defined here. Apart from that, if a derived class uses at least one of the types from the template header then it becomes a templated class as well at which point all the above rules apply to the derived class as well. Also, one thing to note is that we must explicitly pass the templated arguments at the time of creating a templated class instance since the types are ambiguous to the compiler by just looking at the arguments passed to the constructor.

Moreover, I learned what specialization is and how to specialize a member function of a templated class to process a particular type. Specialization is a technique that helps programmer provide an explicit type for template arguments rather than using a generic type for a function. The reason for this *(if a programmer chooses to specialize a templated class or function)* would be, he/she can instruct compiler how to interpret and work with that template argument type. Specialized functions have a slightly different syntax when it comes to declaring and defining a function. First, a template header for a specialized function must be empty *(for full template specialization)* like so ‘’template <>” followed by the function definition “Foo<const char\*, const char\*>::Foo() {}” which in its prototype instructs compiler that if a Foo object is created that contains two const char\* as its template argument type, it should execute this function rather than generic version of constructor. We can talk a little bit about ‘operator+=()’ specialization in this workshop which will help me convey my message about why we need to specialize certain functions. The PairSummable class takes two templated arguments and while the operation *(concatenate “, “ between two strings)* might work for std::string type passed to PairSummable at the time of instantiation, it certainly would not work for fundamental types like int, if we were to generate output like so in this workshop where we adding the ticket sales *(its underlying type is int)*. If the templated type for the value is std::string we would not have any issues to concatenate a string but when it comes to lets say working with templated type, int, it will break the code since we cannot concatenate “, “ to a type that is int. So for that reason we must treat that template initialization differently and hence use the template specialization technique for our code to run our application smoothly if a template type happens to be a std::sting which will generate correct output.

Apart from that, the workshop instructions ask us to provide the reason for “what *= initial* in the prototype means”. It provides a default value to the value attribute in PairSummable so that it can be instantiated with a default value if it is not provided to 2 argument constructor, otherwise utilize that value to initialize the value attribute in PairSummable class if provided. If PairSummable is instantiated with int type, the value of value attribute would be 0, and for string type, value of value attribute would be null terminated string.

Also, the reason why we don't need to specialize the initial value for the summation is that PairSummable is instantiated with 2 different types as its templated arguments in main, once with type PairSummable<std::string, std::string>, and another PairSummable<int, std::string>. As noted in this reflection, templated arguments are considered as part of the template definition that is generated by compiler at the time of compile time which means that instances generated with PairSummable<int, std::string> and PairSummable<std::string, std::string> are completely different types. When working with PairSummable<std::string, std::string> instances’ summation attribute *(its type is V which happens to either be a std::string or int in this workshop)* gets initialized to empty string, which is perfect for us since we must concatenate the values of subset per set in a collection of sets of key-value pairs and then specialized += operator takes care of utilizing string operations like concatenation for instance variable, summation unlike PairSummable<int, std::string> version where instance variable, summation gets initialized to 0 and then += operator uses the arithmetic operations on instance variable, summation which is the favorable behaviour in this case.

Furthermore, I learnt about non-type template parameter. Non-type template parameter are parameters whose types are declared explicitly within template header, making them non-generic type. Their best use case would be as we saw in this workshop where we passed an unsigned int at the time of template instantiation so that we can create a statically allocated array of collections which act as a maximum size allowed for an array. Non-type parameters must be const, constexpr or a prvalue since their value must be known at compile time because we are working with templates which get compiled at compile time.

Lastly, the only difficulty that I faced in this workshop was to concatenate “, “ for a value of filtered key as we loop over the collection. The logic flows like this: local instance of type T(PairSummable) is created in accumulate function of SetSummable class which calls += operator overload defined in PairSummable where we are to perform this concatenation. To go about solving this challenge, I created a static variable local to that += operator overload function which holds the value of the key to current instance. Since static variables remain in memory for the existence of the whole program I checked to see if the key is changed, update the local static variable to the value of key and set the summation to empty, null terminated string. Setting the summation to empty string helps me reset my summation to an empty string so that I don’t end up with a concatenated string from previous key’s values *(summation is a static variable as well which will hold all the values until the existence of the program which why it is necessary to reset it for every dissimilar key).* Once I figured that out += overload for PairSummable<int, std::string> was easy since it was almost similar logic to that of += operator overload for PairSummable<std::string, std::string>.