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

To begin with, I learned 3 new function syntax where we can write our logic just as normal as any given function, but they give us flexibility and applicability which is why they are very popular and a go-to choice for C++ programmers these days.

First, I learned function pointers. Function pointers already existed in legacy code like C programs and just like any other C language functionality they got carried over to C++ as well. Function pointers are pointers to functions that store the address of an instruction for a function stored in memory. Their declaration goes like this, “bool (\*asc) (int a, int b)”. The round brackets are necessary since they identify that a pointer to a function has been declared. If we don’t put the round brackets, then it becomes just a normal function that return bool \*. Also, the return type and the arguments that this function pointer receives is obvious from its declaration. A function pointer will only hold an address to an identical prototype function. If the prototypes mismatch, then undefined behaviour can happen. We can also create statically allocated array of function pointers like this, “bool (\*sort[2]) (int a, int b)” and then we can assign different addresses to those 2 elements in a function pointer. To assign an address to a function pointer we simply put the name of function on right hand side of assignment and to call the function stored in function pointer, we simply use the identifier of function pointer, followed by function call operator and then passing in necessary arguments in function call operator. Function pointers are to be used when we have somewhat complex logic inside a function, and it depends on nothing but function parameters. Function pointers are also referred to as callback functions. Callback functions are functions that are passed into another function as an argument or assigned an address altogether, which are then invoked inside a function later to complete a routine or action depending on a condition if any. A live example can be found in this workshop where we are to assign an address of client defined function, “bookAddedObserver” & “movieAddedObserver” to a private member(function pointer) in the class which is then called back later once the private member(function pointer) is assigned an address in “setObserver” function.

Second, I learnt about functors. Functors are classes that have the function call operator overloaded. It is a class just like any other in C++ which can be inherited, can perform polymorphism(parametric or inclusion), among other things. We usually create a function object of a functor so that we can use () operator on that object to call the overloaded () operator. Functors are used when we have more complex logic, and we need to maintain a state in private members of functor. For example, in this workshop we had to maintain a state(goodWords, badWords & how many times badWords were replaced) in SpellChecker class and based on that state we performed replacement to the string that we receive as a parameter in () operator overload if any of the words(finding badWords in string text) in the string are misspelled.

Thirds, I learnt about lambda expressions. Lambdas are anonymous functions that are internally a functor generated by the compiler. Since they are generated by the compiler, there is no way a programmer would know its type which is why we use keyword auto when we create a lambda. The implementation of a lambda is “auto lambda = [](){}”. Square brackets indicate capture list if any. If we want to capture any variables that are defined in local scope above its definition, we can do so by specifying “=”, “&” or variable name among other combinations. “=” captures all local variables by value which means we cannot update a captured variable and “&” captures all local variables by reference and allows us to modify the captured variable. And due to its capture list, they are also known as closures since they capture all the local variables that are define within same scope. The round brackets include any arguments that are to be passed to the lambda at the time of making a call to it. It is good practise to be as restrictive as possible when declaring arguments. If the arguments are not to be changed, we should make them const and to avoid unnecessary copying, declare them as references. The curly brackets are implementation of lambda, and all logic goes into to these brackets. Lambdas are used for usually a one-time use, which contain simple logic which are only required locally. An example can be found in main where we define a lambda and is used just a few times, locally within main.

Reflection explanation points:

* SpellChecker constructor generates an error and to handle the error I am currently using exception logic provided by C++. However, from my perspective, I can think of performing error handling by creating public member function that returns current state of an object. Then, I can query that function from main to find whether an object is in safe empty state. If it is in a safe empty state, I know that an error occurred while performing some operations on the data members of the class. However, there is a major flaw to this design. I as a programmer am not sure what kind of error appeared. I sure can create a member array of strings that specifically store what errors appeared and then query them, but client cannot know right away that an error occurred, we cannot proceed further. Currently we are only performing error handling if file name passed on command line does not exist, but if we expand the logic in constructor later and it turns out we need to handle more errors too, then this design would not work. On the other hand, with the help of exception logic I can report the error right away to the client that an error occurred and depending on the program we can either proceed to execute the logic or end the program by informing client an error occurred including the type of error.
* The best way that I see right now to not duplicate Movie and Book logic is to inherit common functions from a base class that implements them and then make Movie and Book a derived class so that they both don’t contain any duplicated logic, except for the dissimilar functions that are implementation dependant on the respective derived classes.
* When a custom constructor is declared and defined, that constructor takes over(removes) the compiler provided default constructor, which means any time we create an object of type Book or Movie we must call the custom constructor since a default constructor is not present. However, if we make the custom constructor accept default values in its declaration, then at the time of creating the object if no parameters are passed to that custom constructor, then it acts as a default constructor for our class at which point, we don’t need an explicitly defined default constructor.