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

In this workshop I learned the functionality provided by STL(Standard Template Library) and the convenience of using STL contents. First, I learned containers, like Vector and List which are templated classes and provide various functionality when it comes to storing a collection of data. Built-in types, array are statically allocated containers which cannot be increased in size, which is what Vector and List provides a C++ programmer by implementing dynamic allocation for the data they store.

We can store data into a Vector by using a push\_back function provided by the library which will add any data added passed to it as arguments to the end of the Vector. It has numerous other functions like insert(to insert a given data at a given position) including multiple overloads for each function provided by the library. Vector in its implementation has a pointer of type T that holds address to the first element, and as we add data to it, its size grows. Since it is a dynamically allocated array assigned to a pointer, data is stored contiguously in memory and because of its pointer implementation it is very fast to access a given data stored within a vector, however it is slow in terms of inserting data since it needs to resize its data member every time we remove or insert a data into it.

On the other hand, List also referred to as Doubly Linked List which means we can add/remove data to the both the ends of a List and remove/add data anywhere in the middle of the List. It is very fast to insert/remove elements from a List however its slow when traversing through a List due to its implementation. A list usually has a structure where each object of a structure has 3 data members, 2 pointers named prev and next and an attribute of type T(Implementation of STL list can be complex than what I am explaining here). Each object holds a pointer to previous object and a pointer to next object. Let’s an object named Alpha is created who is a head of the list then its next pointer will point to an object called Charlie. Alpha’s prev pointer can be nullptr at the moment. Charlie’s prev pointer will point to Alpha. Now let’s say we want to insert Beta into the current linked list. It is quite simple to insert or remove something into a List. To do this, Alpha’s next should now point to beta, Beta’s prev should point to alpha. Then Charlie’s prev should point to Beta and Beta’s next should point to Charlie. There are other things to consider other than above mentioned steps (*for example creating a temporary object that holds Charlie’s details(prev, next & data) so that we don’t lose Charlie in the process of breaking the link between Alpha and Charlie which will then be used to make a connection between Beta and Charlie*), but the steps give a generalized idea of how Lists work. And due to this implementation, it is very fast to insert/remove elements(*to remove we do the same procedure but the other way around*) into a list, however slow to traverse through a list. Also, lists don’t store data contiguously in memory which is also another reason why its slower to traverse through it. List just like Vector has a huge list of functions along with multiple overloads to do almost anything we want with a container.

I also learned Iterators that are part of each templated container in STL. Iterators are essentially objects of class Iterator, (depending on container since Iterators have different categories) which help traverse through containers of STL. Iterators can be thought of as pointers since they have \* and -> operators overloaded which can help us give access to the underlying data stored at a given position. The most common use of iterators is when looping over a container since they have 2 functions namely begin and end that return iterators, where begin will return an iterator that points to first element in a container and end will return an iterator that points to one past the last element in a container.

Also, vectors only deallocate memory for its data members. They are not responsible for deallocation when it comes to client-side data if that data is allocated dynamically. Any dynamic memory that is created on client side/programmer side needs to be deallocated by the client/programmer since vectors do not keep a track of dynamic memory that is assigned to them. For example, let’s say vector has a pointer data member called m\_data which is given type int\* at the time of instantiating a vector object. Now we create 5 dynamically allocated objects of type int in main.cpp and insert them into vector, at that moment vector will dynamically allocate memory for 5 objects of type int\* and copy the addresses of client provided dynamically allocated data into m\_data. At the time of destruction, Vector class’s destructor will only delete the memory that was allocated for m\_data and memory that m\_data was pointing to which was dynamic still stays in memory waiting to be freed, which is why we need to deallocate any dynamic memory that we pass into a vector (the implementation for vector would be way more complex than just m\_data, but it is to a general idea of why vectors don’t deallocate memory for programmer side code).

To detect what went wrong in terms if an invalid data was found in a record or an unknown type of vehicle was extracted we can return a value from scoped enum called Errors that, for example for a vehicle other than Car and Racecar we can return Errors::InvalidCarFound, in case of a file I/O error like EOF we can return Errors::EndOfFileReached, in terms of invalid data extracted for example we were expecting a double but got a char, then return Errors::InvalidDataFound, in case of incorrect condition we return Errors::InvalidConditonOfCar. Client has access to the enum so that they can query to see which error has occurred. We might have to do some extra work to deallocate any memory allocated before returning the errors when working with enum style error handling since try catch throw clean any resources automatically by calling destructor on the local object created before a throw is encountered.

For the current Inheritance hierarchy(Interface), we must use pointers to Vehicle type since Vehicle is an abstract base class. If we pass the Vehicle type to vector instead of Vehicle\*, when vector creates an object, the compiler will throw an error, cannot create instance of abstract base class, which is why pointers are used when we are to create polymorphic objects in a hierarchy. Also creating pointers of polymorphic types are helpful in situations like deciding which functions to call depending on an object’s actual type rather than its declared type since calling the functions on a declared type/early binding will give us unexpected results, like compile time or runtime errors depending on situation. Thus, late binding/dynamic dispatch is used in an inheritance hierarchy which implements an Interface and concrete classes are inherited from an Interface, which is implemented using pure virtual functions and creating a pointer for polymorphic objects.