Name: Soham Thaker Sid: 011-748-159 Email: sdthaker1@mysneca.ca

To begin with, I learned is rule of 5 in this workshop. Last term we learned about rule of 3 but this term we expanded that rule of 3 to 5 by including move operations. To understand move operations, we first need to understand lvalue and rvalue references. Any object that is about to die before the program ends, for example, the object is temporary, any value that can only appear on the right-hand side of the assignment, a variable or an object that was defined within function scope are all qualified as rvalue reference. A rvalue reference is denoted as ‘&&’. On the other hand, lvalue reference as those references who survive even after a scope ends or are not temporary. They can appear on left or right side of an assignment operation. A lvalue reference is denoted as ‘&’.

Once we’ve known the difference between rvalue and lvalue references we can explain the move operations and the rule of 5 in better way. The rule of 5 now includes move constructor and move assignment operator also called as move operations. Both accept parameter of same type however two major differences between the argument types of copy operations and move operations are, 1) the argument instance in move operations is not constant, since we are going to modify the argument instance inside these special functions, & 2) the move operations accept a rvalue reference denoted as ‘&&’ rather than lvalue reference, which denoted as ‘&’.

There are specific steps defined for move operations that we must follow to not cause a memory leak or a crash and successfully move the state of argument object to the current object. We can also define move operations as moving the ownership from one object to another rather than copying the data from one object to another. As far as move constructor’s semantics are concerned, we can simply make a call to the move assignment operator by writing ‘\*this = std::move(other)’. And as far as move assignment operator‘s semantics are concerned, we first check for self assignment, then we perform shallow copy on non-resource members from argument instance to current instance, followed by deallocating the current instance’s memory, then assigning the address of argument instance’s resources to current instance and then changing the state of argument instance’s non-resource members to safe empty state and changing the state of resource members to nullptr, ending it with returning reference to current instance. Changing the resource members’ state to nullptr is very important and not setting it to nullptr in move assignment operator will cause crash of our application. The reason for this is that when the move assignment operator ends, destructor is called for the argument instance, which will delete the dynamic memory allocated to it leaving the current instance’s resources as dangling pointers. Now the crash/undefined behaviour will occur when we try to access the state of current instance outside of move operations since they both shared the same memory because we assigned the address of argument instance resources to current instance’s resources in move operations and data at that memory is deleted which can possibly cause segmentation fault or read access violation.

In certain situations, like if an object is about to die, for example, if it is defined within a function scope and the return value of that function happens to by value, compiler will automatically call move operations on an application if the move operations are defined to improve performance. The 2nd point mentioned in 2nd paragraph helps compiler figure out whether to do performance improvements and call move operations if the object is near its end of life.

C++ also provides us with a function defined in utility header which helps us convert/cast a lvalue reference to rvalue reference if we as a programmer decide that we want to move the ownership of object to another object or we decide if the object is not needed anymore. The function is called as ‘std::move()’ which under the hood casts a lvalue reference to rvalue reference which will help compiler figure out to make a call to move operations if they are defined in an application.

The main reason for the significant time difference between copy and move operations in this workshop is that in move operations we just steal the resources which means we are only copying a 4-byte memory address from one pointer to another pointer and this operation happens very fast rather than copy operations where we one by one copy all the state of resources in a loop from one resource to another resource which is performance and execution time intensive.

Apart from that, I also learned how to find the time difference between start of an operation and end of an operation by using std::chrono library. We can use a ‘std::chrono::steady\_clock’ data member class called ‘time\_point’ that can store a given time point. We usually assign it the current system time by calling the member function of steady clock which is called as ‘now()’ which returns an object of type ‘time\_point’. To calculate the difference between time points we first need to define an object of type duration which is a class defined within ‘std::chrono::steady\_clock’. Then we use a non-member function of duration class called ‘duration\_cast’ which accepts a typedef as its template argument which specifies units of time for example, nanoseconds. Then we pass the duration\_cast the object of type time\_point, usually start and an end by performing a subtraction operation at which point it will return an object of type ‘std::chrono::steady\_clock::duration’ to be stored back into a duration object. From that point we can call count member function on a duration object which will return count of ticks in units of time. The value of units of time would be the one which was passed as templated argument to ‘duration\_cast’ at the time of casting 2 different ‘time\_point’.