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

Firstly, I learnt what function bindings are and the concept of virtual functions. The compiler binds a function call based on the object’s type. The type of the object determines the member function to call in an inheritance hierarchy. Usually, the binding of a member function has either of the 2 forms: 1. Early binding, which is based on the static type of the object, & 2. Dynamic dispatch, which happens based on the object’s dynamic type. If we declare an object as “Container B”, here the static type of this object is Container and compiler will bind all the calls for any functions on the object B at compile time. However, if we declare B as a pointer and initialize it dynamically, like, “Container\* B = new Shape” and let us say Shape is a base class whereas Container is a derived class, now B points to the address of Shape class, thereby an object to the Shape class and gets initialized with the Shape part of the class. If we have a function called display in both these classes which is non-virtual and if we call this function in the client code, on B, for example “B->display()”, this will call the display based on the static type of B, which is Container, so in these circumstances we might not get the result as expected where we have to make it a polymorphic object which means making the display function virtual or any function in that regard where we want to postpone the binding of the function at run time. We just need to declare it as virtual in Base class and rest of the functions with same identifier will be virtual as well. Once display is declared as virtual, compiler will postpone the function call, “B->display()” at run time and it will evaluate the address stored in that pointer which in this case would be of Shape and thus calls display from Shape object.

Furthermore, I learned why we make destructor of classes in a hierarchy virtual. The same method as explained about dynamic dispatch above applies to the virtual destructors. Over the course of time, polymorphic objects change their addresses where they might be pointing to different classes in a hierarchy and for that reason, compiler must call the right destructor once an object goes out of scope otherwise memory leak can happen. Same example as above, we have 2 classes, Shape(Base) and Container(Derived) now if we create, “Shape\* S = new Container” and if we don’t declare the destructor to be virtual, then at the time of destruction of the object S, it will only run the Shape’s destructor because that’s its static type of the object, however if we declare the destructor as virtual then, it will run the Container’s destructor and then run the Shape’s destructor since compiler evaluates the address stored at S on run time.

Moreover, I also learned what are pure virtual functions, abstract base classes, and concrete classes. Pure(refers to lack of any implementation) virtual functions are functions who are not implemented in a class and if at least one of the functions are not implemented in a class then that class becomes an abstract base class where we cannot create an object or instantiate an object(thus compiler forcing the programmer to implement it later on in the program) of an abstract base class, however we can still create a pointer or a reference to that object where it will be instantiated with a class that is called as a Concrete class, which essentially means that all the pure virtual functions have been implemented and we can create an instance of that class. We say that a function is a pure virtual if we declare it as “virtual display() = 0” which means the function will be implemented later in a class hierarchy. The concept of dynamic dispatch also applies to the pure virtual functions as well where compiler will do a late binding depending on the poly objects’ dynamic type, it will call the respective function.

Besides, I also learnt what are Interfaces and why they are used. Interfaces are classes where all the functions as pure virtual functions and do not have any data members. Interfaces are just an abstract design of what is to come next when we design a class hierarchy. We can say that an abstract base class is a base class that defines an interface whereas a concrete class is a derived class that implements the interface and gives a specific meaning to the interface.