Task 1

We will build a small class hierarchy and experiment with how it works. Begin by creating the base class of the hierarchy. Call it *myLocation*, and have it encapsulate two data members which are integer variables for the *x* and *y* coordinates in the plane. It should also have three member functions: a no argument constructor, a two argument constructor, and a *display* function. Note that having two functions with the same name, the two constructors, which we have called "overloading" the function, is a form of polymorphism. We will see another form of polymorphism later in the recitation.

To help the exploration, place the following print statements inside the three member functions: "In the no argument constructor of myLocation", "In the two argument constructor of myLocation", "In the display function of myLocation".

In menu option 1 show that the two constructors and the display function work.

Task 2

Now we will derive a class from our base class. The new class will be called *myPoint*. The difference between *myPoint* and *myLocation* is that *myPoint* adds a data member for a color. For the moment, you can let this be a character variable. It will also have a no argument constructor, a three argument constructor, and a *display* function.

We will need to change the access keyword for the data members of *myLocation* from private to protected. Actually, since we said we were going to create a class hierarchy, we could have done that already. Yes, you could leave them as private, but that would force you to add a lot of extra code to this recitation. You can use this simpler style for today.

In menu option 2 show that the two constructors and the display function work for *myPoint*. But wait! Where is the output for the constructor of *myLocation* coming from? Since *myPoint* is derived from *myLocation*, we have placed a call to the constructor of *myLocation* in the member initialization list of the constructor for *myPoint*. The member initialization list is executed before the body of the constructor for *myPoint*.

Task 3

In menu option 3 we will use our class hierarchy in a slightly different way. Again, create and display two *myPoint* objects, one with each constructor. Now declare *loc* to be a pointer to the base class. Assign the address of the first derived object to *loc*. Then display what *loc* points to. Do it again for the second derived object.

Note that what you see is the display function of the base class, not of the derived class. For one thing, you don't see the color of the points. You have also managed to assign an address of an object of one type to a pointer to an object of another type ... why didn't that cause some kind of type mismatch error? Because of the class hierarchy. You are allowed to use derived classes in most places where base classes are expected.

Task 4

Now we will make a small change to the base class and will see that the output of menu option 3 changes as well. In the *myLocation* class, place the keyword *virtual* in front of the display function. Then rerun the program and choose option 3. This time the same two calls to display the derived objects pointed to by *loc* show the output of the derived objects instead of the base object. This is possible due to late binding. That is, the compiler does not know which implementation of the display function will be used. It is determined at run-time from the object instance.

Task 5

Up to now, all the data structures that you have used (arrays, dynamic arrays, lists, linked lists, and the STL containers) have contained elements of a single data type. These are called "homogeneous" containers. However, it is often useful to group different types of data into a single container, that is, to create "non-homogeneous" containers. Consider extending our class hierarchy beyond the *myPoint* and *myLocation* classes. We can add *myCircle*, *mySquare*, *myRectangle*, etc. We can use objects of these classes to draw a figure, perhaps to draw objects of our *Person* class. If we can place all of these objects into a single container, then we can traverse the container with a simple loop and display all of the elements with a single call to the *display* function.

We could demonstrate the concept with our current class hierarchy, but it will make more sense if we at least add a *myCircle* class to our class hierarchy. Do that now. A circle is really just a "fat point", which we can model by adding the single data member, *diameter*. Let one of the constructors have four arguments, and one have no arguments.

Now, do the following. First, declare *theArray* to be an array of pointers to *myLocation*. You can let the size of the array be 6. Next, declare two objects of each of the class types ... two *myLocation* objects, two *myPoint* objects, and two *myCircle* objects. Then, assign the addresses of each of these objects to an element of *theArray*. Finally, use a simple for-loop to display all of the elements of the array.

Task 6

Well, maybe you think that was a little hard. Can we simplify that code? Who needs those stinking pointers? Copy what you did in case 4 to case 5. Change the declaration of the array so that it contains myLocation objects instead of pointers to myLocation objects, simply remove the *. Remember we said above that derived classes could often

be used whereever base classes were used. Also the six assignment statements to theArray look easier to write without the address of, &, so remove those. So we are placing the objects into the array, instead of the addresses of the objects. Finally, change the statement in the loop ... it no longer needs the arrow, it just needs the dot operator. Compile your code. Everything looks good, right? Now run your code ... what happened?