It is time to learn more about the power of polymorphism with class inheritance using abstract classes with pure virtual functions.

We need to update the `MobileRobot` class, update the `Tracked` class, and implement the `QuadRotor` class.

This assignment is worth a total of 10 points:

- Up to 6 points for successful compilation and correctness,
- Up to 3 points for correctly following the Google style guide, and
- Up to 1 point for correct documentation.

# Directory

We will continue using the following directory structure:

- **bin**: To contain executable binary files.
- **build**: To hold the intermediary .o object files.
- **include**: All .h header files will be located here.
- **lib**: A directory of useful classes for your project, provided by me. Normally this directory would hold true library archives which would get compiled into linkable libraries. This is outside of the scope of this class. I have provided you a makefile for this directory and you may make all the .o files you need by running "make library" from either the root directory `04hw/`, or the lib directory `04hw/lib/`.
- **src**: All .cc source files will be located here.
- **test**: Directory to hold the unit test source files which you will, hopefully, write to test your classes before they are submitted.

# MobileRobot:

Instances of the `MobileRobot` class represent something that can move, without describing how it moves. We need to update and implement the following:

- **Constructor(s)**: All mobile robots must know their max speed; therefore, `MobileRobot` shall have a default constructor which assumes speed is 0.0 units/time and a constructor which accepts speed as a double precision floating point value. The constructor must assert that speed is non-negative. These may be the same constructor. In addition to assigning a non-negative speed, the constructor shall also provide the new instance a unique name.
- **Destructor**: After a robot falls out of scope, its `id_` shall no longer be non-unique. This method should be virtual so that if a `MobileRobot` pointer is deleted, the child's destructor will be called.
- **Accessor**: Because different mobile robots may have different max speeds, create an accessor named `speed` which returns the mobile robot instance's speed as a floating point value. This method *should* promise the compiler it does not change any members' value.

- **Accessor**: Mobile robots need some means by which to communicate if they are to coordinate or take direction from a controller. This means they must each have a unique name. Create a method called `id` which returns an instance's unique name as a string from the C++ string library. The method *should* return the name as an r-val and promise the compiler it will not change members' values.
- **Mutator**: A user may wish to assign a name to a robot. The method `id` shall accept a string from the C++ string library and return true if the string represents a unique name, false otherwise. If the name is unique, the mutator shall assign the string to the robot instance's `id_` member variable. Do not forget to set the robot's previous name as non-unique.
- **Accessor**: Given the new class hierarchy of the `Point` class, we can now maintain the `location_` in the `MobileRobot` class and should move the accessor here. The method's id should be `location` and return value should be an r-val pointer to a `Coordinate` and should promise not to change the calling object.
- **CanTranslateTo**: With the new class hierarchy of the `Point` class, the `CanTranslateTo` method can be made pure virtual and moved to the class. Its parameter should be changed to an unmodifiable `Coordinate` pointer.
- **Translate**: This method accepts an unmodifiable `Offset` pointer and attempts to move the robot in that direction and for that distance, to the limit of its speed.
- **IsUnique**: A good user might want to check whether a name is unique for attempting to assign it to a robot instance. Provide users of your class with a **static** method named **IsUnique** which accepts a string from the C++ string library and returns true if the string has not yet been assigned to a robot.

# Tracked:

The `Tracked` class now implements a virtual parent and so we must be mindful to implement all virtual functions if was want to instantiate objects of the class. In addition, we must modify the class to make use of the new `Point` and `Vector` class hierarchies. Note that `Tracked` will only use two-dimensional geometry.

- **Constructor(s)**: Tracked robots need to know their location and shall make no assumptions about such. There is no default constructor for a Tracked Robot. There are two ways to provide location to a Tracked instance,

  1. one uses a `Coordinate` pointer from the `lib/include/point.h` library and
  2. the other uses a pair of double precision floating points.

  In either form, an additional double may be provided to set some initial speed. If no initial speed is provided, it is 0.0. Be mindful of the way in which `location_` is managed. You should create and destroy memory to copy the value(s) given in the destructor.
- **Destructor**: In addition to the work done in a parent constructor, the memory for the `location_` must be managed.
- **CanTranslateTo**: Implement this method as described in the parent class.
- **Translate**: Implement this method as described in the parent class.

# QuadRotor:

The `QuadRotor` class implements a virtual parent and so we must be mindful to implement all virtual functions if we want to instantiate objects of the class.

Cont.

- **Constructor(s)**: Tracked robots need to know their location and shall make no assumptions about such. There is no default constructor for a Tracked Robot. There are two ways to provide location to a Tracked instance,

  1. one uses a `Coordinate` pointer from the `lib/include/point.h` library and
  2. the other uses a trio of double precision floating points.

  In either form, an additional double may be provided to set some initial speed. If no initial speed is provided, it is 0.0. Be mindful of the way in which `location_` is managed. You should create and destroy memory to copy the value(s) given in the destructor.

- **Destructor**: In addition to the work done in a parent constructor, the memory for the `location_` must be managed.

- **CanTranslateTo**: Implement this method as described in the parent class.

- **Translate**: Implement this method as described in the parent class.

# Submission

: To get credit, you must upload a zipped (not tarred, gzipped, or 7z) archive containing your submission files in the the directory tree provided. Given the difficulty I have seen with so many student unable to successfully create and archive files, you will archive **ONLY** the directory structure which I provided you.

Late submissions will be handled as per syllabus. I am attempting to fit four more assignments into this semester so that struggling students have a chance to bring up their grade; consequently, there will be no extensions.

The End.