Project 5
Cruising into Design Patterns and Smart Pointers
The first part of a two-part project.

Due: Friday, April 2, 2010, 11:59 PM

Note: The Corrections & Clarifications posted on the project web site are officially part of these project specifications. You should check them at least at the beginning of each work session.

Because this project has a fairly open design, these specifications contain only few specifics of how your code should be written, and so run a risk of being under-detailed. Be sure to start early enough to have an opportunity to ask for a clarification or correction.

Purpose

This project is a extension/elaboration/refinement of Project 4. You will need a working version of Project 4 as a "starter." This project will provide a foundation for the final project, and so is actually the first part of a two-part project. In this project you will:

- Demonstrate the value of a good architectural framework by making a small change with big effects on future extensions.
- Try out smart pointers to automate memory management using the to-be-Standard TR1 smart pointers if possible, or a supplied intrusive reference-counting smart-pointer template if not.
- Make use of the Singleton pattern to make Project 4's Model globally accessible in a well-controlled way.
- Demonstrate the value of an OOP framework by adding a new derived class of Ship, observing how only the code required to support the new class needs to be added, while the remaining client and other code is unmodified. As the gurus say, "inherit in order to be reused;" "add features by adding code, not by modifying code."
- Apply the Model-View-Controller pattern in complete form by implementing additional kinds of views.
- Get additional practice on using predefined concrete classes for mathematical computation.
- Get additional practice designing your own classes.

This project assignment attempts to leave as much of the design under your control as possible. Thus the specifications will be considerably less detailed than before, and will emphasize how the program should behave, and very little about how you should accomplish it. Where necessary to make sure you work with the informative design possibilities, some design constraints are specified - a few things you may or may not do in your design. While the design is under your control, you are expected to make good use of the OOP concepts and guidelines presented thus far in the course. While you are free to modify Project 4 as needed, you should not have to make any substantial changes at all to the Sim_object class hierarchy. Rather, you will be adding to this hierarchy, designing a new class hierarchy for View, and modifying Model and Controller to accommodate these. In other words, your Project 5 solution should be clearly based on Project 4's. The final project will be a further extension of this project. Be sure to study the Evaluation section below for information on how the code quality will be assessed.

The specifications are expressed as steps in the order you should do them for maximum benefit and smoothest sailing in this project.

Step 1. Make Cruisers defend themselves.

If you have a good architecture, then many changes can be very easy to make. It should be clear that we could change the internal behavior or parameters of the ship classes without any effects on the rest of the program. What about more impressive changes that involve the public interface? These will also often be easy to make in a good architecture because a good division of responsibilities between classes and their member functions results in very compartmentalized code. So interesting things can be done with only a few small changes. Demonstrate this by making a small change that enables much more interesting behavior on the part of Ships.

Change Ship::receive_hit so that it takes a second parameter, a pointer to the Ship making the attack, but for now, the function does not use this parameter. Change Warship::fire_at_target so that it supplies "this" as the second argument when it calls receive_hit on its target. Your program should rebuild successfully; note that because we have changed the public interface of Ship in Ship.h, most of the other components will have to be recompiled. At this point the Ships should behave the same as before.

Now have Cruiser override receive_hit to first call Ship::receive_hit, and then if this Cruiser is not already attacking, it calls Warship's attack function to tell itself to start attacking its attacker. A Cruiser will now automatically counter-attack if it is attacked.

We now have the ability to make Ships do all kinds of interesting things based on knowing who their attackers are - instead of counter-attacking, they could do something else, like running away in the right direction.

Step 2. Use smart pointers for all Sim_object family objects, and sink Ships faster.

This change is pervasive in the code. In the real world, we probably would have started with smart pointers from the beginning. Change all containers and pointer variables used to refer to Sim_objects, Islands, Ships, etc., over to smart pointers using the Standard TR1 smart pointer classes (see the Example Code directory for examples of their use with gcc 4). If you are not using gcc 4, you can either download and install the Boost TR1 library for e.g. MSVS, or use the Smart_Pointer template, Smart_Pointer.h, in the course Example Code directory.

Hint: See if your IDE allows you to simply do a global search/replace throughout all of the project source files of "Ship *" with "Smart_Pointer<Ship>" or "shared_ptr<Ship>", etc. This makes the change-over very easy.

Once you have switched over to smart pointers, you should not have any "raw" pointers for Sim_object family objects anywhere in your program, nor should you have any explicit deletes of these objects - the smart pointers should do this automatically, and containers of smart pointers will destroy the contained smart pointers when they get destroyed, eventually automatically deleting all the pointed-to objects.

Once you have the smart pointers in place, it is now possible to simplify how ships disappear when they are sunk; Project 4 avoided a dangling pointer problem by having the ships go through a series of states to ensure that any other ship (e.g. an attacker) had a chance to disconnect its pointers before the sunken ship was deleted. However, with smart pointers used throughout, there is no need for a long painful sinking process. Each component simply discards its pointer when it should, and the object is automagically deleted when it is no longer referred to.

Make the following changes to the specifications and code from Project 4:

- Remove the Ship::is_on_the_bottom() function. The project 4 function Ship::is_afloat() is all we need; it returns true if the Ship is in one of the Afloat states; false if not.
- Change Ship::update from Project 4, replacing the update of the sinking status with the following:

  First check whether the ship is sinking or not:

  > If this Ship is still afloat, check the resistance. If the resistance is greater than zero, go to the update of the afloat state below - we aren't sinking. If the resistance is less than 0, set the state to Sunk, output "sunk", and do nothing further in this or any future update. Note that the state will not change in the future.
- Change the description of Ship::describe from Project 4, replacing item #2 with:

  > 2. If the Ship is Sunk, output "sunk" and describe nothing further.
- Remove the other sinking states as possible values for the ship state and any outputs that formerly depended on them.
- Then, change Model::update so that after updating everything, it checks for ships in the Sunk state (that is, not Afloat), and removes (erases) the smart pointers to them from the containers. If there are no other smart pointers referring to a Sunk ship at that point, it should get automatically destroyed.
- Change Warship::update so that if its target is no longer afloat, or no longer exists, it terminates its attack and discards its pointer to the target. If this is the last smart pointer keeping the target "alive", this is when it will be when it gets deleted.
- Because Warship won't "forget" its target until it updates, change Warship::describe() to output the supplied new "absent" message in case your Warship is in Attacking state when its target no longer exists or is sunk.

Cruisers should be keeping track of their targets through some type of smart pointer. But if two Cruisers are attacking each other, they will be referring to each other, and thus will keep each other "alive" even if they both sink, or the program is quit. The same thing might happen if they refer to each other indirectly, such as through another Cruiser - this is called a "cycle" in graph theory. A similar problem might keep Islands alive if in a future version Islands refer to each other or to Ships that refer back to the Island.

You will have to devise a solution for making sure that the cycles get broken so that all objects will get destroyed at the most appropriate time, so that for example in this project, two Cruisers that are attacking each other get deleted if a "quit" command is issued. You only have to solve cycle problems that are potentially present in this version of the program. For example, you can assume that Islands will not participate in cycle problems. But you must handle cycles in a way that will simplify solving any future cycle problems that might arise in future extensions or elaborations of the program. In other words, your solution must work in a way that allows it to be easily extended to any additional sub-types of Sim_objects. The specifics of a good solution will be different depending on whether you are using the TR1 smart pointers (very easy, if somewhat verbose) or the Smart_Pointer.h template (requiring a bit more thought).

Once you do this, your program should run identically as before, but you will see differences in the order of destruction of Sim_objects, based on the details of which containers get emptied when, or which smart pointers get destroyed. I suggest leaving the destructor messages in as long as possible so that you can observe this interesting "garbage collection" at work, and verify what happens when you add the remaining program features.

Step 3. Make Model a Singleton class.

Once you complete this step, your program should run identically as before, but now it will be possible to do Step 4, which requires program-wide access to information from the Model. While you are messing with Model, create another Island in Model's constructor, to be "Treasure_Island" at (50, 5), 100 tons of fuel, production rate of 5.

Step 4. Add a Cruise_ship class.

A Cruise_ship has the capability to automatically visit all of the islands in a leisurely fashion. It behaves like a normal ship until you tell it to go to an island with the "destination" command. It then announces that a cruise is starting and then it visits the specified destination island, then visits the closest next island, followed by the island closest to that, and so forth, visiting each island only once. It travels at the speed specified in the initial command. When it arrives at an island, it docks, and then it stays for a few updates. On the first one, it attempts to refuel from the island. On the next update it does nothing while the passengers see the local sights. On the third update, it sets course for its next destination (the closest unvisited island; in case of a tie, the first in alphabetical order). When it has visited the last island, it returns to the first island, the one it was originally sent to. When it arrives there, it docks, does not try to refuel, and announces that the cruise is over.

If during a cruise, the Cruise_ship is given any navigational commands (stop, course, position, or destination), it announces that the cruise is canceled, forgets where it has been, and follows the command. If the command is another destination command to an Island, it will start a new cruise. A Cruise_ship can be created using the create command, like the other kinds of ships.

You will need to make a small addition to Model so that the Island information is available. Do not try to compute the route in advance. Rather, at each Island, simply compute the next Island to visit, go there, and repeat.

Specific data: A Cruise_ship has fuel capacity and initial amount 500 tons, maximum speed 15., fuel consumption 2 tons/nm, and resistance 0.

Design constraints: You are not allowed to modify the public or protected interface of Ship to add the Cruise_ship class; you have to work through the available virtual function interface defined by Ship. You may not use any downcasts, safe or not. The idea is to demonstrate software "plug and play:" take maximum advantage of the existing Ship family structure and commands, so that Cruise_ship is implemented only by adding the Cruise_ship class and adding a couple of lines to the ship factory code.

Error handling. The design requirement here is to take maximum advantage of the public interface of Ship, and you are not allowed to change this interface in order to implement Cruise_ship. Since the Ship movement functions can throw exceptions, your Cruise_ship code should call them in such a way that an error will not leave your Cruise_ship in an invalid state. For example, if a Ship movement function is called to set the destination position and speed, but the supplied speed is not possible, that Ship function will throw an exception. To avoid a peculiar state, call the Ship functions prior to setting the Cruise_ship state and outputting the various Cruise_ship messages.

Clumsy implementation and peculiar side effect: Since you have to work through the public and protected interface provided by Ship (specified by Project 4), the only way a Cruise_ship can tell whether it has been sent to an island is if a specified destination position matches the position of an Island. Fortunately, all islands have small integer-valued coordinates, so floating point issues are unlikely to confuse the matter. If you write the obvious code for this, the side effect is that a "position" command can also start a cruise if it supplies exactly the position of an Island.

This kludginess is a strong suggestion that Project 4 didn't quite represent the domain like it should have - Islands are "first class" places for Ships to have as destinations, so Ship should have been given the ability to understand directly that an Island is a destination and go to that Island, as opposed to the just a position that happens to be an Island. Redesign, anyone? (Not for this project!)

Step 5. Apply Model-View-Controller and extend the View class to different kinds of views.

This step is an example of a common activity during software development. It is discovered that it would be a good idea to have variations on a capability already present. In this case, the View from Project 4 is just fine, but we want another kind of View as well, and the ability to have more than one View simultaneously active. In addition to modifying Model to handle any number of view objects, this requires "refactoring" the View class from Project 4 in some way. We want to retain the basic Model-View-Controller (MVC) logic in Project 4, but simply supply additional kinds of Views, and make them very flexible and under the control of the program's user. The overall structure will be very similar to how GUI code is normally organized, making this project a much better example of the MVC pattern and how it is used. Thus, your code should follow the presented pattern.

The main difference between this use of MVC and normal GUI MVC is that normally in a GUI, when Model updates views, each View immediately redraws itself, while here, the View just "remembers" what it was told, and then draws itself when the user issues a "show" command. It works this way in this project (and the previous) because if our dumb text-graphics Views always drew themselves after updating, they would generate megabytes of output, which would just get in the way.

The Project 4 View, called a *map view* in this project, will still be available; the user can still control its origin, scale, and size. You can recycle the code with little modification.

The new kind of view is a *bridge view*. See the samples. It shows what things look like if one is on the bridge of a particular ship and looking straight out the front, over the bow of the ship. Islands and other ships will be seen ahead and to the sides, depending on where the ship is and what direction it is going in. Where another object appears from our left to our right corresponds to the angle it has from the bow of our ship.

Terminology. Our own ship, that we are standing on, is called *ownship*. The direction our ship is pointed in, or is going, is its *heading*, expressed as a compass angle (0 degrees is North, etc.). The *bearing* of another ship is the compass angle from ownship to the other ship, regardless of what direction ownship is going. For example, suppose we are going North-West (heading is 315) and another ship is North-East of us - its bearing is 45 degrees, because it is North-East, which is a compass angle of 45 degrees. The *angle on the bow* of the other ship is the difference between our heading and the other ship's bearing, in this example, 90 degrees. That is, if our ship is heading North-West, and we are standing on the bridge looking straight over the bow of the ship, we would have to look 90 degrees to the right to see the other ship. Thus, an angle on the bow of 0 degrees means the other ship or island is straight ahead, -45 degrees means it is to the left, -90, fully on the left, and +90 degrees means that it is fully to the right. Angles more extreme than +/- 90 degrees mean the other object is behind us, +/- 180 degrees is directly behind us.

Specific data: The bridge view ranges from -90 to +90, with 0 in the middle. The size and scale are fixed at these values. If an object is more than 20 nm away, it is not shown, so objects can appear as we get closer. To avoid a mess when docked at an island, if an object is less than 0.005 nm in distance, it is not shown on the display. This prevents an Island or another Ship from appearing if we are in the same location. If two objects would appear in the same cell of the display, a '**' is shown in that cell.

The character graphics matrix is 19 by 3 cells, with 2 characters/cell. The 3 rows contain the bottom row, used to show Ships or Islands, while the top two rows are just decoration - they represent the sky, or in case the Ship is sunk, the surrounding water.

Each horizontal cell represents 10 degrees. The leftmost cell represents angles in the -80s, the rightmost cell, angles in the +90s. That is, -90 is the most negative value that appears on the x-axis labels, and it appears as the most negative value in the leftmost cell of the view. The rightmost label is +90, but if the plotting computation is done analogously to the map view, +90 is the smallest value that will appear in the rightmost cell, which actually goes up to, but not including, +100. The goal is to ensure that -90 to +90 would be visible and appear as labels when, for simplicity, the computations are done with the same logic as the map view. However, this simple approach actually makes the displayed range larger on the right than the left.

Computations: Construct a Compass_position using ownship's position as the first constructor argument and the other object's position as the second. This will contain the bearing between ownship and the other object, and also give you the range (distance) of the other object from ownship. Note that if two objects are at the same location, the bearing is indeterminate, so the .005 rule keeps this from being an issue. Compute the angle on the bow as the bearing minus this ship's heading. If the result is less than -180 degrees, add 360.0 to it to "wrap" it the other way. Likewise, if the result is greater than +180 degrees, subtract 360.0 from it to wrap it the other way. Use the same sort of computation to compute the horizontal subscript as in Project 4's View. The value corresponding to the "origin" is -90.0 degrees, and the scale is 10.0. Be sure to test this view using various bearings and headings - for example, make sure you have used ownship and the other object in the right place in the computations; reversing them will be obvious only at certain angles.

Multiple bridge views are a feature: You can have a separate bridge view for every ship. You create one by opening a bridge view and providing the ship name. You get rid it of closing the one corresponding to a ship name. You can do this at any time, for any combination of ships. In this project, there is no need to have more than one map view, but it can also be opened and closed at any time.

Updating and commands. If a view is open then it gets updated by Model after every **go** command, as in Project 4. The **show** command outputs all of the open views by calling their draw() functions. The user can **open** a view to cause the desired view to be created, updated, and displayed (by the **show** command) and then **close** the view to get rid of it.

This gives the following new commands that the user can issue:

open_map_view - create and open the map view. The Project 4 view commands size, zoom, and pan control this view if it is open. Error: map view is already open.

close_map_view - close and destroy the map view. Error: no map view is open.

Note: The other map view commands for scale, origin, size throw an Error if no map view is open.

open_bridge_view <shipname> - create and open a bridge view that shows the view from the bridge of ship <shipname>. Error: bridge view is already open for that ship.

close_bridge_view <shipname> - close and destroy the view for the ship. Error: bridge view is not open for that ship name. Note: It is not necessary to look up the Ship of that name; the name should be enough to identify the bridge view.

The order in which views are output is the order in which they were last opened. (Project 4's pervasive alphabetical order rule does not apply to this.) For example, if the user opens a bridge view for Valdez, then the map view, then a bridge view for Ajax, they will be displayed by the show command in the order: Valdez, map, Ajax. If the Valdez view is closed, then of course it is no longer output. But if the user opens a bridge view for Valdez again, then the order of display will be map view, Ajax , Valdez. The bridge view for a sunken ship displays a "water" pattern shown in the sample output (alternating "w' and '-' showing that the bridge of that ship is now under water) and a modified heading that says the ship is sunk. The map view or the view from a ship that is still afloat does not show sunken ships, as in Project 4.

Design constraints: Your solution must conform to the following constraints:

1. You will need to add some additional accessor functions (or possibly using declarations) to the public interface of Ship in order to implement the bridge view because you will need some of the information from Track_base to compute the information shown in the bridge view.
2. The two kinds of view (map view and bridge view) must be represented by *two classes*, as opposed to some kind of "mode" of a single class.
3. Additional kinds of views must fit in easily in the view class structure you design, and your design should follow the guidelines presented in the course for good class design.
4. Your design must have the property that a new kind of view can be added to the program without changing Model, or its relationship to the view classes, in any way at all. In other words, your views must be a good fit for the Model-View-Controller pattern.
5. When a view is opened, a view object must be created with new; when closed, the object is destroyed.
6. You should use smart pointers with the views so that all your memory management is now automated - no deletes should be present anywhere in the program except for the ones provided by the smart pointer classes. The Singleton Model object would be the only exception - using a smart pointer for a Singleton makes no sense (meditate on it if you are tempted to do so).

Step 5. Remove the constructor/destructor messages.

You should keep these messages in your program until the very end, to make sure that everything is being destroyed when it should be. Just remove or comment them out before submitting your project. As mentioned before, the order in which objects get destroyed

will be rather different from Project 4, but every object that gets created should eventually be destroyed, at the earliest, when it is no longer needed, and at the latest, when the program is terminated.

Files to submit

Submit the same set of files that you did for Project 4 except as follows:

- You must use the supplied **p5_main.cpp** instead of p4_main.cpp.
- **Cruise_ship.h, .cpp**.
- **Views.h, .cpp** (notice the plural!) instead of View.h, .cpp. All of your code for the views (both bridge views and map views) must be in this .h/.cpp pair
- At your option, you can include a header file called something like "Smart_ptr_config.h" to make it easier to move between MSVS and gcc (see below) and/or define typedefs for your smart pointer types.

The autograder will supply the Smart_Pointer.h identical to the one in the course Examples directory; the TR1 smart pointers will be available automatically on the autograder platform (which uses gcc) as long as they are accessed in the way shown in the Examples directory.

General Requirements

To practice the concepts and techniques in the course, your project must be programmed as follows:

1. The program must be coded only in Standard C++.
2. You must follow the recommendations and correctly apply the concepts presented in this course.
3. Your use of the Standard Library should be straightforward and idiomatic, along the lines presented in the books, lectures, and posted examples.
4. You must use the smart pointers correctly, following the guidelines and concepts for their use. For example, a design problem in the supplied Smart_Pointer.h Reference_Counted_Object class is that the increment and decrement functions are publicly available, but should not be called by anything else but the Smart_Pointer template code.

Project Evaluation

This will be the last autograded project in the course. Because the design is more open, and you are allowed to modify public interfaces to the limited specified extent, component tests will not be performed. The autograder will test your program's output to see whether your Cruise_ship and views behave according to the specifications.

In Project 6, the second phase, you will be choosing features to implement and designing how to implement them, so you will supply some design documentation and demonstration scripts along with your final source code. Project 6 will thus not be autograded, but human-graded throughout. The course schedule simply does not make it possible to complete a Project 5 design evaluation in time for you to respond to it with Project 6. Thus, you will get an autograder score only on Project 5, and then Project 6 will fully human-graded, but it will be evaluated for both the Project 5 design concepts and the Project 6 design requirements. For this reason, no instructor's solution for Project 5 will be posted. You will have to build your Project 6 based on your version of Project 5. This arrangement is somewhat awkward, but past experience is that it works. One thing that makes it work is the following nasty warning:

Spot Check Warning: I will check that the required components and design concepts appear to be present in Project 5, and some easy-to-check aspects of code quality and good design have been done; if not, your autograder score for Project 5 will be substantially reduced. In other words, you should design and write Project 5 well so that you can then focus on the design problems in Project 6. If you do a poor job on Project 5, you will be sorry when you do Project 6.

The code and design quality for both the Project 5 and Project 6 components will be weighed extremely heavily in the final project evaluation. It is absolutely critical not only that you design and write your code carefully, but also that you take the time to review and revise your code and your design, and ensure that your final version is the best that you can do. This care needs to cover both the Project 5 and Project 6 work.

Suggestions for Using TR1 on different platforms

TR1, not being Standard yet, has to be included in different ways on different platforms. This is a place where conditional compilation with the preprocessor is useful. You can use #if to determine what platform your code is currently running on, and then #include the appropriate headers or declare different things such as different typedefs. For example, the following detects whether you are using MSVS by seeing if MS's preprocessor symbol for the compiler version number is defined, and if so, #includes MS's headers for part of tr1; otherwise the gcc header is included. The inverse can be done by using

```
#ifdef _MSC_VER // other possibilities are WIN32 _WIN32 or _WIN64
#include <functional>
#else
#include <tr1/functional>
#endif
```

Alternatively, you can detect the presence of gcc with a different preprocessor symbol and include those headers:

```
#ifdef __GNUC__  // leading and trailing double underscores
#include <tr1/functional>
etc.
```