Project 4
Sailing into Object-Oriented Programming

Due: Friday, March 19, 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 of the amount of detail required for this project, these specifications are more prone to errors and omissions than normally. Be sure to start the project early enough to have an opportunity to ask for a clarification or correction.

Purpose

In this project you will practice the following basic OOP techniques:

- Working with a set of classes that have clearly defined roles and responsibilities, using a simple form of the Model-View-Controller pattern.
- Working with a class hierarchy that uses virtual functions to achieve polymorphic behavior.
- Using abstract base classes to specify the interface to derived classes.
- Observing the order of construction and destruction in a class hierarchy.
- Using a "fat interface" to control derived class objects polymorphically.
- Calling base class functions from derived class functions to re-use code within the class hierarchy
- Using mixin multiple inheritance to take advance of a pre-existing base class.
- Using private inheritance to provide re-use of implementation rather than interface.
- Decoupling classes as fully as possible by minimizing header file dependencies and using a simple form of Factory.

In addition, you will practice some additional programming techniques:

- Working with objects that have state and state changes.
- Further use of exception-based error handling for user input errors to simplify the program structure.
- Further use of the Standard Library fully to simplify coding.

Finally, certain aspects of the coding may be new to you, although they are ancient and traditional issues:

- Using floating point numbers and dealing with some of the representational issues. For example, comparisons for equality are usually unreliable.
- Using a 3 dimensional array-type data structure.

Overview

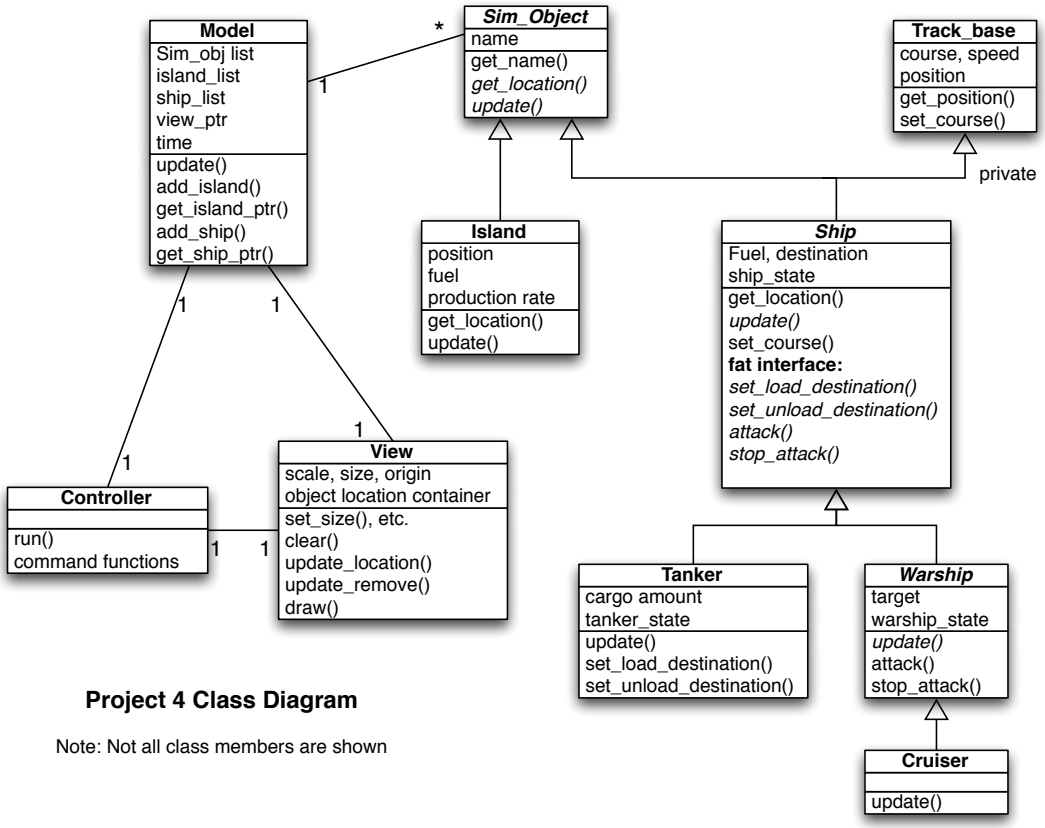
The program implements a simulation similar in spirit to many computer games and actual simulation applications in which simulated persons, vehicles, ships, etc. move around and behave in various ways. The user enters commands to tell the objects what to do, and they behave in simulated time. Simulated time advances one "tick" or unit at a time; in this project, a "tick" is one hour of simulated time. Time is "frozen" while the user enters commands, and then the user commands the program to "go" to advance one tick of time.

The world in this simulation consists of a limitless "ocean" that contains islands, and ships can sail around in the ocean from one place to another. The program can display a grid that represents a map-like view of the two-dimensional world. There are two kinds of objects, Islands and Ships. Islands are ports that have a supply of fuel, and some of which can produce fuel. Ships can move about, consuming fuel, and can dock at and refuel at an Island. If a ship runs out of fuel, it stops and can move no more (a condition called "dead in the water"). There are two kinds of Ships: Tankers haul fuel oil between islands, working automatically once given their orders, and Cruisers can shoot at other Ships and sink them. The user (you) can command a ship to move around, and to dock and refuel at an island, or if appropriate to the Ship, haul oil or attack. You can display the status of the ships and islands, or view the map, to help you choose what commands to issue.

Later projects will have fancier possibilities. This project essentially is building a framework in which much more complicated simulations can be constructed easily in future projects. This is a fairly complex project because there are many classes involved, and a relatively large number of separate header and implementation files. However, thanks to the use of OOP techniques, relatively little code has to be written. The basic design of the program is specified, since learning how to construct such frameworks works best from examples. Later projects will be more open in design. The navigation in this domain is also somewhat complex, but the use of a pre-existing library of classes and functions eliminates the need to work out most of the math and its coding.

Basic Architecture of the Project

The architecture of the project is summarized in the Class Diagram shown below. To keep the diagram from being insanely detailed, only some representative members are shown for each class. The Model-View-Controller (MVC) design pattern shows up here in a simplified form. It helps organize the top level of the program into a structure where the responsibilities are divided up between three objects: The Controller object interacts with the human user, handling all the input, and controls the rest of the program. The Model object keeps track of the simulated "world" of ships and islands; it is the keeper of which ships and islands exist, and keeps them all up



Project 4 Class Diagram

Note: Not all class members are shown

to date, and instructs them to describe themselves either to the console output or to the View. The View in this case prepares and presents a map-like view of the World.

A key feature of object-oriented design shows up here: these three objects are very isolated from the details of the Ships and the islands. Basically, the Model and Controller do not know about the different kinds of Ships; they only know about the base class Ship, not about Tankers and Cruisers. They know that Ships and Islands are both a kind of object in the world, called Sim_objects: Ships and Islands are both Sim_objects. View doesn't even know this much; it only knows how to plot names in locations on a map - it doesn't know anything about what the names refer to. New ships are created by a Factory, which allows the Controller to create a new Ship on command without having to incorporate any knowledge of the different ship types. Because of this structure, in the future new kinds of Ships or Islands can be added with little or no change to the Model, View, or Controller. The Sim_object class hierarchy is designed to accept new kinds of Ship and Island sub-classes.

Basic Inheritance Structure

The class Sim_object is an abstract base class that describes what all of the simulated objects in the world do - it has a name, you can get its location, and you can tell it to describe itself or update itself. There are two derived classes: Islands and Ships. Islands store fuel oil, and can produce it at a certain rate (possibly zero).

Ships can be told to move on courses, go to specific islands or positions on the ocean, or refuel from an island. If they run out of fuel, they become immobile. If they get hit by too much gunfire, they sink. Ships are quite complex because they have some complicated movement behavior, but derived classes of Ships simply inherit this behavior, and can override it to give more interesting patterns. For example, Tankers can move fuel from one island to another, and can be told where to load and unload fuel, and will then start shuttling between the two Islands, working independently thereafter. They use their Ship capability to move around. Cruisers are an unaggressive form of Warship; they can be told to attack another Ship, and if the other Ship is in range, they will start firing at it, but will stop as soon as the target is either sunk or moves out of range. The framework is in place to easily install a variety of Ship types, each with their own specialized behavior, because the Ship and Warship base classes already provide much of the functionality required. Ship provides a "fat interface" to allow the specialized functions of derived classes to be accessed by the client code. In this case, all attempts to tell the wrong kind of Ship to do something are user errors and treated accordingly.

Finally, the implementation of Ship is simplified because it multiply-inherits not just from Sim_object, but also Track_base, a pre-existing base class recycled from a Navy simulation project. Track_base, and the class and function libraries it includes, Geometry and Navigation, provides the functionality needed to "Navigate" - send ships moving about on courses in terms of standard maritime concepts, such as compass headings, figure out how far away things are from each other, etc. A Track_base object has a position, a course, a speed, and even an altitude. It was used to describe "Tracks" - which is Navy jargon for anything that can be tracked by radar and sonar - ranging from aircraft to submarines. Here, the altitude of a Track_base object is always zero, corresponding to a surface ship.

Polymorphism magic

The Sim_object class declares a virtual update function and a virtual describe function. Each class overrides these functions to provide definitions appropriate to that type of object. When Model tells each object to update itself, each object "automagically" does the right thing for its type. Tankers continue moving oil, Cruisers continue an attack, Islands produce more oil, and so forth. The virtual functions in the Sim_object base class allow the client code to treat the objects identically, while each object does its own behavior.

However, it is not possible to treat all objects identically. This project uses two techniques to treat different kinds of Sim_objects differently depending on what type of object they are. The first technique, *separate containers*, enables Controller to be able to tell whether a name refers to a Ship or an Island. This is done simply by having two different containers in Model, one for Ships, the other for Islands. When objects are created, we know what types they are, and simply put them into the appropriate container. We can tell whether a name is an Island simply by searching for the name in the Island container. We can put all the objects into a third Sim_object container to use when we treat them as all the same, such as telling each one to update itself.

The second technique is the *fat interface*. Different kinds of Ships have different possible behaviors; Tankers cannot attack other Ships, and Warships can't transport oil. How do we command a Ship to do these things unless we know what exact kind of a Ship it is? The answer is that the Ship class declares virtual functions for the *union* of functions we want to call on the derived classes - thus Ship declares virtual functions for attacking and transporting oil, even though the Ship class does neither of these. The resulting collection of virtual functions is "fat" because it is this union of derived-class capabilities. Then, Tanker overrides the oil transporting functions, and Warships override the attacking functions. The Ship class provides "default" versions of these functions that simply throw a "Can't do that!" Error. If the user, working through Controller, tells a Tanker to attack they get the error message; if they tell a Cruiser to transport oil, they get the error. If they tell a Ship to do something that particular Ship can do, then it happens; otherwise, they get the Error message. Thanks to polymorphism magic, the Controller doesn't have to know anything about the different kinds of Ships for this to work.

Decoupling

The combination of fat interface, separate containers, and factory function means that different parts of the program can be very independent, or decoupled, from each other. The key is that each component does not #include any unnecessary header files of any other component. As elaborated more below, Model and Controller don't have to know about the different kinds of Ships, only Ships. The Ship_factory component shields Controller even further by providing a way for Controller to request the creation of new kinds of Ships without knowing anything about them. In fact, Tanker.h needs to be #included in *only two* places: Tanker.cpp and Ship_factory.cpp. Likewise Cruiser.h is #included only in Cruiser.cpp and Ship_factory.cpp. This means that the Tanker or Cruiser behavior can be changed with no impact on the rest of the program. Furthermore, a new kind of ship can be added to the program by adding only the .h and .cpp for the new type, and modifying Ship_factory.cpp to create one of the new type on request. The entire rest of the program modules will work without modification - they don't even have to be recompiled! Now that's decoupling!

The Navigation Domain

This domain of ship navigation is has its own traditions, and one of the goals of good programming is to honor the terminology and practices of the application domain.

- Real ships sail around on a sphere, with their positions being kept in latitude and longitude coordinates whose correspondence to distance depends on where on the globe you are. Working in these terms requires spherical trigonometry, which is only approximate because the Earth is not a perfect sphere. There is a much simpler approach, called "plane sailing" in navigation, and it is accepted as reasonably accurate over relatively small distances (e.g. a couple hundred nautical miles). It is used in "plotting" - keeping track of where a ship is relative to other ships and to nearby land. Plane sailing is used in this project: ships are assumed to sail around in a limitless flat ocean, the real plane, with coordinates measured in units of nautical miles (nm, a nautical mile = 6080.2 feet or 1852 meters, 1 min of arc along the equator - not really important here!). The origin of the plane is arbitrary; there is no standard location on the planet for the (0, 0) point.
- We assume that the map is viewed in the normal way, North at the top, South at the bottom, East on the right, and West on the left. In terms of normal Cartesian (x,y) coordinates, North is assumed to be the positive y direction, South in the negative y direction, East in the positive x direction, and West in the negative x direction.
- The direction a ship is sailing in is normally expressed in terms of a "heading" - degrees on the compass: 0 degrees is North, 90 degrees is East, 180 is South, 270 is West. The heading of 360 is always converted to 0, and values less than zero or greater than 360 to the corresponding angles within the [0, 360) range. So to sail from (x, y) position (10, 20) to position (0, 10) involves sailing on a heading of 225 degrees (South-West) for about 14.14 nm (the square root of 200). When a ship is in motion, its state is usually described as being on a certain course (a compass heading) and at a certain speed.
- We'll assume there are no winds or currents; so the direction a ship goes depends only on what heading it is steered in.
- The Std. Lib. math library has a variety of trigonometric and other functions, but these functions assume the normal mathematical relationship between angles and (x,y) coordinates. In terms of map directions described above, these functions assume that East is 0 degrees, North is 90 degrees, West is 180, and South is 270. Furthermore, the math library functions follow a standard computing practice of using radians rather than degrees. Thus, positive angles correspond to counter-clockwise movement from 0 to 180 degrees, and negative angles to clockwise movement from 0 to 180 degrees. Fortunately, the Geometry and Navigation modules provided make the appropriate transformations between math radians, math degrees, and ship compass headings. Plan to examine what is available in these modules before writing code to do any of these computations - they can be tricky to get right.
- A major simplification has been made in this simulation: Ships do not collide with each other or with islands. That is, you can assume that if a ship and island (or another ship) are in the same location at the same time, they actually invisibly scoot around

each other - another way to put it is that they have zero size. Detecting collisions is a major technical puzzle in computer game programming, and involves complications that aren't relevant to our concerns in this course.

Floating-point Issues

Computer scientists often get into trouble with floating-point numbers, because unlike integers, floating-point is not an exact representation, but only an approximate one. Throughout this project, double-precision is used, following the normal practice when any significant computation will be done, such as trigonometry. The key consequence of the approximate nature of floating-point is that two floating-point values that are supposed to be mathematically equal will almost never be equal at the bit-by-bit level that the == operator tests for. About the only time one can be sure they will compare exactly equal is if the mathematics dictates that all the values involved in a computation will always be integral (like exactly 5.0), within the precision of the representation, and have the same bit-by-bit format. Another case is that two values will probably test exactly equal if one has been assigned to the other and both have been stored in memory, as opposed to one being in a temporary floating-point register that might still contain more bits of value (common on the Intel architecture). Needless to say, you can't be sure of this relationship, especially if the compiler does any optimizing.

The approximate values of floating-point numbers can be especially problematic if they are converted to quantized values in some way, such as a discrete state (e.g. "full" tank of fuel versus "non full"), or an integer value (e.g. for plotting on a discrete grid). A difference in the zillionth bit to the right of the "binary point" in the representation can "flip" the quantized result to one value or another, and this zillionth bit's value can depend on the microscopic details of a computation in combination with the CPU architecture and the compiler code generation policy. The overall effect is that in borderline cases, the result of a quantization can appear almost random to a user.

That said, how will I autograde the output of your program? First, some of the key computational code will be provided, and you should use it as-is (or at most with variable name changes). This will ensure that these computations are performed in exactly the same way on the autograder machine. Second, some of the algorithms will be spelled out, so that state changes should happen at the same time. Third, all of the numerical output written to cout will be rounded to two decimal places. This is done by a couple of lines of code in the p4_main.cpp file that is provided. The code for View will have to change these settings, and will be responsible for restoring them (see the handout on Output formatting). While rounding is a quantization operation, because of the great many bits carried by double precision, rounding to two decimal places suppresses almost all of the possible glitches. There may be a few surprises, such as a value of "-0.00", which can result when a small negative number is rounded off, but the autograder knows that this should be considered the same as "0.0". Fourth, certain comparisons will be done with a range, rather than testing for equality, and some values will be assigned (e.g. the amount for a full tank of fuel) to suppress any lingering differences. These will be specified where needed. Fifth, I will attempt to use test cases where borderline situations do not appear.

Finally, I will be careful to check that failures to match on numerical grounds will not be penalized if the code was written in the specified way, and will re-evaluate test cases if they appear to hinge on borderline cases where quantization will cause distortions. Past experience suggests that these problems will be rare. They are worth the risk because it makes it possible to do much more interesting and realistic things in the projects. Later projects will involve less emphasis on exact output matching, and more on code design, so the risk of grading problems will get less.

State Machines for Behavior

The Ships in this project behave in complicated ways. A good way to represent complicated behavior is with a *state machine*. Each object is in an initial state. When commanded to do something, it changes into another state corresponding to the command. On each update, the current state is tested, and the appropriate actions done depending on what the current state is. One of the actions might be to change the state. At most one state change will happen on an update. When the object is next updated, the new current state will control its behavior. Another command, such as **stop**, might change the state as well, whereupon the next update will deal with that new state. An excellent way to code a state machine simply and clearly in C or C++ is to use an enum variable to represent the state, and then simply switch on that variable in the update function, and put the state-dependent code in the cases of the switch. The default case in the switch can be used to recognize an unknown state due to programming errors.

In this project, the objects can have multiple states, one for each class they are comprised of. The Ship class has a state that represents whether the ship is moving, stopped, docked, and various ways of being unable to move, such as out of fuel or sinking. If a Ship is attacked and begins sinking, it first becomes Sinking, and then when next updated, becomes Sunk, and then On the Bottom. If a Ship is On the Bottom, Model will remove and delete it at the end of the current update process. The reason for these multiple states is that all of the objects are referred to by pointers, and following a "dangling" pointer to a deleted object produces undefined results. The chain of states thus make it possible for an Ship, upon being updated, to interrogate another Ship, and if it is no longer Afloat, to cease interacting with it or referring to it before it is deleted.

In addition to the Ship state, Tankers and Warships have their own set of states and state changes, concerned with moving oil and attacking respectively. The rule is that to update a Tanker or Warship, we first update their Ship state, and then update their Tanker or Warship state. This update can make use of information about the Ship state - such as whether a Tanker is still Afloat or is moving.

Class Responsibilities, Collaborations, and Dependencies

Take special note of the responsibilities listed for each component and class. These are critical to the design - in a good OO design, each class has well-defined and limited responsibilities that make sense in the application domain. Confusing and difficult designs result when it is not clear what each class is responsible for. Understanding the responsibilities will help you know what the code is supposed to do, and where different information is created, stored, and used.

Also specified are the couplings - the dependencies between the files in different components. This can help you prepare a makefile, but it also aids understanding of the design - minimum coupling of components is highly desirable, both because it cuts down on recompilation time, and makes it easier to modify one component without having the changes propagate to other components. Dependencies on Standard Library components (e.g. std::string) are not described. Good header file design is required in this project; consult the handout on the topic.

The course server site contains files for the components listed below, either complete files, or "skeleton" files where you must supply the missing content. The comments in these files provide additional specification information.

In all cases, you are not allowed to change the public and protected interface for any class. Except where specified, the private members are for you to choose. When a particular function definition is supplied, you must use it in your code in the specified way.

Important: Students new to inheritance often fail to take advantage of it. In this project, derived classes have many opportunities to use functions in a base class to get their work done. If you finding yourself writing code in a derived class that is similar to code you wrote for a base class, you are doing something wrong. Stop! Sort it out - ask for clarification or help!

Geometry.h, .cpp, Navigation.h, .cpp, Track_base.h, .cpp, p4_main.cpp

These components are supplied on the course file server, and should be used as is, without any modification. Their responsibilities are:

- Geometry is responsible for representing points, coordinate systems, and vectors in the real plane, and providing classes and operators for computing these.
- Navigation provides classes and functions for doing "plane sailing" in the navigation domain.
- Track_base represents an object that can move in the navigation domain. It is responsible for maintaining information about the position of an object, its current course and speed, and computing a new position for the object given a specified amount of elapsed time.
- The main module, p4_main.cpp, simply creates and starts the Model-View-Controller framework.

Track_base depends on Geometry and Navigation. The main module depends only on the Model,View, and Controller components.

Sim_object.h (no .cpp)

This class is an abstract base class that describes the interface, members, and capabilities of all of the simulated objects in our simulated world. Each derived class is required to override the get_position, describe and update functions to provide their own specific behavior. Because it is basically an interface class, its only direct responsibility is maintaining the name of the object, which is a std::string.

Sim_object depends on Geometry.h because of the need for a returned Point object to represent a position. It has no other dependencies.

Island.h, .cpp

An Island is a kind of Sim_object, so it inherits from Sim_object. Its responsibilities are:

- To maintain its position, which once created, cannot be changed, and to supply it upon request.
- To describe itself upon request. It outputs its name, position, and the amount of fuel available.
- To maintain the amount of fuel oil available at the Island. This is done as follows:
  - The initial amount is specified when the Island is created, along with a production rate.
  - There is no limit to how much fuel can be stored, so any amount is accepted from an outside source (such as a Tanker).
  - Some other object can request an amount of fuel oil, and the island will either return that amount, or the amount currently available, whichever is less, and subtracts it from the amount available. The fuel oil can be used either as cargo or as fuel for a ship - the island doesn't care. It outputs how much fuel it delivered in response to the request.
  - When updated, an Island adds the amount of oil produced in a unit time to the amount available, and outputs the current amount - it does this only if the production rate is greater than zero.

Island.h, .cpp depends only on its base, Sim_object. It knows nothing about, and has no dependencies on, any other class.

Ship.h, .cpp

Ship is the most complex class in the program, because Ships are complicated and this class represents all that Ships have in common. Some of this capability it has through its inheritance from Track_base. Since Track_base is used only to implement Ship, Ship privately inherits from Track_base. That is, none of Track_base's public interface is relevant to the rest of the program, so it becomes private to Ship. While Ship has many detailed responsibilities, they all pertain to the state of a Ship:

- To maintain its position, which changes as a result of Ship motion, and to supply it upon request.
- To maintain its state, and supply it upon request, either through a limited public interface, and an additional protected interface for the use of derived classes. Details are provided below. Aspects of the state are:
  - Whether the ship is afloat or is in the process of sinking.
  - How much fuel is available.
  - What kind of motion the ship is engaged in (e.g. on an open-ended course or to a destination), and whether motion is possible.
  - The current resistance of the ship - this is reduced when the ship is hit with gunfire, and determines whether the ship starts to sink.
- To implement commands for moving (or stopping), docking, refueling. If these commands cannot be executed by this particular Ship due to its state, an exception is thrown.

- To interact with Islands for docking and refueling.
- When updated, the Ship updates its state depending on its current resistance, whether it is afloat, the kind of motion, and the amount of fuel, and outputs some information about its current state. Details are described separately below.
- To describe itself, the Ship outputs its name, its position, and then specific information about its state. To save on bits, only the most relevant information for the state is shown - e.g. if the ship is sinking, we don't care how much fuel it has aboard. See the samples and detailed specifications below.
- To provide a fat interface for derived classes. This is a set of virtual functions for commands accepted and acted upon only by derived classes. Currently, these are functions are setting loading and unloading destinations, and starting and stopping attacks. The functions are overridden by derived classes as appropriate for the class. In the Ship class, these functions have a "default" implementation that throws an Error exception. Thus if a particular type of Ship is given a command that it cannot execute because it is the wrong type, the default exception-throwing function will execute. Example: A Tanker is told to attack; an Error exception is thrown because Tanker does not define an override for the attack virtual function.

Ship.h depends only the Ship bases, Sim_object and Track_base. Ship.h needs only an incomplete declaration for island. However, Ship.cpp must include Island.h to support docking and refueling.

Tanker.h, .cpp

Tanker is a kind of Ship, and so inherits from Ship, and overrides the fat interface virtual functions appropriately. It also overrides the update and describe functions to add its own capabilities, and calls the base class definitions of these functions to invoke the capabilities common to all Ships. The specific responsibilities of a Tanker are:

- To accept and act upon commands for setting loading and unloading destinations. Details are below.
- When both destinations have been set, it starts running automatically between its two destinations.
- When updated, to maintain its own state. Details are below; in summary, this involves:
  - The amount of cargo and fuel currently aboard.
  - While docked at a loading destination, it attempts to both refuel and fill its cargo hold. Since an Island may not have enough fuel on hand immediately, the Tanker will repeatedly request fuel from the island until the hold is full.
  - When docked at an unloading destination, to provide all of its cargo to the island.
  - After it has completed loading or unloading, it starts moving at full speed to the other destination automatically.
  - Where possible, a Tanker uses the public interface of its Ship part to perform these functions (such as starting to move to its next destination).
- When told to stop, it "forgets" its loading and unloading destinations.
- To describe itself, a Tanker outputs both its type name ("Tanker"), the generic Ship information, and then its own specific additional information, such as whether it is loading.

Tanker.h depends only on its base, Ship; Island can be handled with an incomplete declaration. Tanker.cpp needs to include Island.h to support the loading and unloading functions.

Warship.h, .cpp

Warship is an abstract base class that inherits from Ship, and represents what all Warships would have in common, such as an amount of firepower, and the maximum range over which the firepower can be used (e.g. the target must be close enough for naval guns to reach). It overrides the fat interface virtual functions for attacks. It also overrides the update and describe functions to add its own capabilities, and calls the base class definitions of these functions to invoke the capabilities common to all Ships. The specific responsibilities of a Warship are:

- To represent the firepower and maximum range, and make this information available to derived classes.
- To maintain the state of an attack, and make this available to derived classes.
- To maintain a pointer to the target of an attack, and make it available to derived classes along with whether the target is within range.
- To accept and act upon commands for starting and stopping attacks on other Ships. Details are below.
- When updated, to maintain its own state. The specifics:
  - Stopping the attack and outputting a message if either this Ship or the target Ship is no longer afloat (as defined by Ship::is_afloat()).
  - Outputting a message that the attack is ongoing.
  - Note that the Ship state is updated first, so a Warship can be attacking while it is still moving, for example.
- To describe itself, a Warship outputs the generic Ship information, and then its own specific additional information, such as whether it is attacking.

Warship.h,.cpp depends only on its base, Ship.

Cruiser.h, .cpp

Cruiser is a kind of Warship, and so inherits from Warship. Warship handles many of the details, so there is not much code in this class. The only functions it overrides are the update and describe functions, in which it first calls its base class definitions to invoke the capabilities common to all Warships (and thus to all Ships). The specific responsibilities of a Cruiser are:

- When updated, after invoking Warship::update(), to maintain its own state as follows:
  - If it is attacking, and the target is in range, fire at the target; if the target is out of range, print a message, and stop attacking.
- To describe itself, a Cruiser outputs its type name ("Cruiser") and the generic Warship (and thus Ship) information.

Cruiser depends only on Warship.

Ship_factory.h, .cpp

This component consists only of a single function, a Factory function. A Factory is given some specifications for a particular type of object from a class hierarchy, creates the object, and returns a pointer to it, where the pointer has base class type. The specifications are simple strings and numbers. In this way, the caller can create a desired type of object without having to know, or depend on, the class declarations of the desired object - only the base class definition is needed. There are several ways to implement the Factory design pattern, and this is the simplest. This function returns a Ship * that points to a new Ship object at the specified position, whose exact class is named in a string parameter. For convenience, the string values correspond directly to the class names, but this is not a requirement for a Factory (in fact, C++ has no Standard way to access the programmer-assigned name of a class at run-time). The Ship_factory code interprets the string parameter and then creates the appropriate type of object.

Ship_factory.h only incompletely declares Ship, so any other component can call the factory function without having included the definitions of all the ship types. Ship_factory.cpp must include the headers for all of the ship types that it can create.

Model.h, .cpp

The program creates only one Model object; roughly speaking, it is responsible for keeping track of the simulated world: what the time is, and who the objects are. More specifically, all of the Sim_objects are referred to with pointers, and the Model object is the central repository of these pointers, so that other parts of the program don't have to try to maintain their own lists of pointers and keep them in synch with each other. Thus, Model is the only object that knows what Islands and Ships are in existence. It has the following more specific responsibilities, which are conveniently described as services to the rest of the program.

- Keeps a container of pointers to the all of the Sim_objects, and separate containers for pointers to Ships and Islands. When a new object is created, the pointer to it is given to Model to update its containers; this is done separately for Ships and Islands, so that Model knows whether the new object is a Ship or Island by what container it saves the pointer in.
- A lookup service: Model can be given a name and will find the the pointer to the object of that name in its containers, and return it. If Model discovers that there is nothing of that name, it throws an error exception. The lookup service is provided separately for Ships and Islands.
- A lookup service to see if a name is already in use, so that it won't be used for creating a new object. The Sim_object names must be unique.
- A service to remove a Ship pointer from its containers, and delete the Ship.
- A View updating service: A View is "attached" to the Model with the attach() function, and thereafter, when the Sim_objects change (e.g. a new Ship is added), or get updated, Model tells the View via its update functions to update its map display. Details are below. There is also a detach() function to discard the pointer (set the variable to zero) whose value will become more apparent in the next project.

Since the Model object knows about the Sim_objects that are present, it provides some services for acting on all of the objects:

- A describe service that tells each Sim_object to describe itself, in alphabetical order by object name.
- An update service that increments the time, and then tells all of Sim_objects to update themselves, in alphabetical order by object name. After updating everybody, it then determines which if any ships have been completely sunk (Ship::is_on_the_bottom() returns true), and then removes and deletes them, in alphabetical order by object name. The View is updated along the way (see below for specifics).

Three additional responsibilities:

- Model keeps the master time value for the simulation.
- When the Model object is created, its constructor creates the initial set of Sim_objects and adds them to its containers. This befits its role as keeper of the state of the world.
- When the Model object is destructed, it deletes all of the Sim_objects using the pointers in its containers. As keeper of the world, it is responsible for the final cleanup when the world is destroyed.

Model.h requires only incomplete declarations for Sim_object, Island, Ship, and View. Model.cpp depends only on the Sim_object, Island, Ship, and View classes (no subclasses of these). it also calls Ship_factory from its constructor, and so needs that header file. It absolutely does not depend on the the declarations for Warship. Cruiser, or Tanker, and must not include their header files - it would be a major error to introduce this coupling.

View.h, .cpp

View has a single responsibility, which is to produce the map-like character-graphics display that shows the positions of the objects. Each object is represented in the map by the first two letters of its name. It has a public interface, called by Controller, to change the parameters of the display, such as the map scale. It also has an update_location function, called by Model, to add the information about an object, identified by name, and its location in the map, and a second function to remove an object from the map. Specific details are below. In this program, there is only one View, but logically, multiple views are possible, and will appear in later projects.

View depends only on the Geometry module for its computations.

Controller.h, .cpp

Like Model, there is only one Controller created by the program. It has a single responsibility, to interact with the user, and control the rest of the program in response to user input. Details are provided below, but in summary, the Controller:

- Prompts for, collects, and processes all user input.
- Uses Model services to update objects, or have them describe themselves.
- Uses Model services to look up Ship and Island pointers.
- Calls Ship functions to perform operations commanded by the user.
- Calls Ship_factory to create a new ship.

- Creates a View object and attaches it the model at the beginning of run(), and detaches and deletes it when run terminates.
- Calls View functions to control the View display.
- Detects basic validity errors in user input (see Error handling discussion), while delegating the responsibility for detecting specific errors to the objects with the relevant information (e.g. specific ships).
- Catches Error exceptions thrown in the program and handles them by preparing for new user input (see Error handling discussion).

Controller.h needs only incomplete declarations of Model, View, Ship and Island; depending on your private functions, you may need Geometry for the Point class. Controller.cpp depends on the declarations of Model, View, Ship, Island, and Ship_factory. Both Controller.h and Controller.cpp absolutely must not depend on the declarations for Warship. Cruiser, or Tanker, and must not include their header files - it would be a major error to introduce this coupling.

Utility.h, .cpp

To keep the autograder happy, your project must include these two files even if the .cpp file is empty and does nothing. The supplied .h file contains a simple Error class for use with the exception error handling, which you must use and include in all components that throw or catch errors. You may put other functions or classes of your choice in these files, but only for classes or functions that are used by more than one module - (like the Error class).

Detailed Specifications

Additional details appear in the supplied files. The following is a description of what the key functions do in each class, with attention paid especially to how state changes are done. Compare with the supplied headers and skeleton headers. This section is tediously detailed because the behavior of each class must be exactly specified and implemented if the program is to behave correctly.

Ship Behavior

The possible states are illustrated in the state transition diagram below. This diagram shows the different states, and which states can be moved between, but the conditions for each transition are described in the text below. The top set of states in the diagram are called the *Afloat* states - they are states that a Ship can be in while it is afloat. The lower set are the *non-Afloat* states in which the ship is in the process of sinking. A double-pointing arrow means that the transition can be made in either direction, but a single-pointing arrow means the transition can be made only in that direction. For example, a Ship that is Moving to Position or Moving on Course can run out of fuel and become Dead in the Water, but there is no way to leave this state. However, a Ship that is Moving can be told to become Stopped, and then told to start Moving again. A Ship can start sinking from any of the Afloat states. To save clutter in the diagram, this fact is shown by the label "Any Afloat State" with an arrow pointing to the Sinking state. The ship goes from Sinking to Sunk to On the Bottom. A Ship that is On the Bottom will be removed from the simulation, but this chain of states gives other objects (especially an attacking Warship) an opportunity to disconnect from the sinking Ship in order to avoid dangling pointers, as well as giving the crew time to abandon ship (not currently part of the simulation).

Refer to this diagram while reading the detailed specifics below for how each member function uses the current ship state to determine what to do, and how to change state. Any checks or tests should be performed in the order they are listed in the description of the function.

is_afloat. Return false if the state is Sinking, Sunk, or On the Bottom; true otherwise.

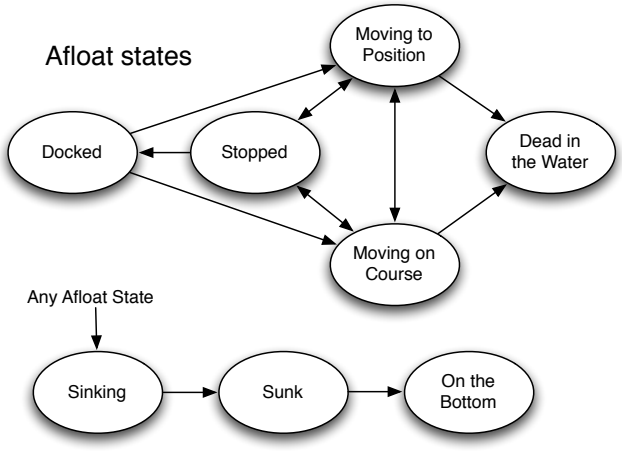
can_move. Return false if the ship is not afloat, or if the ship is Dead in the Water.

is_moving. Return true if the state is Moving to Position or Moving on Course; false otherwise.

is_docked. Return true if the state is Docked; false otherwise.

can_dock. Return true if the ship is Stopped and the supplied island is within 0.1 nm.

get_docked_island. If the ship is Docked, return the pointer to the Island it is docked at; return zero otherwise.



Ship state transition diagram

describe. Do the following if Ship::describe() is called:

1. Output the Ship's name and position. (Note: A subclass will first output its class name (e.g. "Tanker) and then invoke this function)
2. If the Ship is Sinking, Sunk, or On the Bottom, output "sinking", "sunk" or "on the bottom", and describe nothing further.
3. Otherwise, output the amount of fuel, and the current resistance of the ship.
4. Then, depending on the state, output:
   - "Moving to <destination position> on course ..."
   - "Moving on course ..."
   - "Docked at <Island name>"
   - "Stopped"
   - "Dead in the water"

set_destination_position_and_speed. This function causes the ship to start moving to the supplied destination position. Check that the ship can move and that the specified speed <= this Ship's maximum speed. Throw Errors if not. Then use Navigation functions to compute the course (compass heading) and supply it and the speed to Track_base. Output "will sail on ... to" and make the state Moving to Position.

set_course_and_speed. This function causes the ship to start moving on a particular course with no destination position. Make the same checks, and supply the course and speed to Track_base, output "will sail on" and make the state Moving on Course.

stop. Throw an Error if the ship cannot move (it may be Dead in the Water or not Afloat). Set the speed to 0, and output "stopping at", and make the state Stopped.

dock. Throw an error if the ship is not Stopped or is too far away from the island. If it is close enough (with 0.1 nm) set this Ship's position to be equal to the island's position, and make the state Docked. Output "docked at".

refuel. Throw an Error if not docked. Compute how much fuel would be needed to fill up the tank to the maximum capacity. If the amount needed is less than 0.005, set the amount of fuel on hand to the fuel capacity (this resets the fuel tank to be exactly full) and do nothing further. Otherwise, ask the Island to provide the required amount of fuel, and add the result to the tank, and output "now has ... tons of fuel".

receive_hit. Subtract the amount supplied from the resistance, and output "hit with ... resistance now".

update. The actual calculations for the Ship movement are handled by Track_base, and Ship::calculate_movement(), which is provided in the skeleton Ship.cpp file. So the update function behavior specified here has to do with the Ship state and how it is updated. The initial state of the Ship is Stopped and with a full tank of fuel. To update the Ship's state, do the following when Ship::update is called:

First check whether the ship is sinking or not:

1. 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 Sinking, output "sinking", and do nothing further in this update.
2. If this Ship is Sinking, set the state to Sunk, output "sunk", and and do nothing further in this update.
3. If this Ship is Sunk, set the state to On the Bottom, output "on the bottom", and do nothing further in this update.
4. If this Ship is On the Bottom, output "on the bottom", and do nothing further in this update. Note that the state will not change in the future.

If the ship is still afloat, then update the state:

1. If this Ship is Moving on a Course, or Moving to a Position, update the position by calling calculate_movement() and output "now at", and do nothing further in this update.
2. If this Ship is Stopped, output "stopped at <position>" and do nothing further in this update.
3. If this Ship is Docked, output "docked at <island>" and do nothing further in this update.
4. If this Ship is Dead in the Water, output "dead in the water at <position>" and do nothing further in this update.

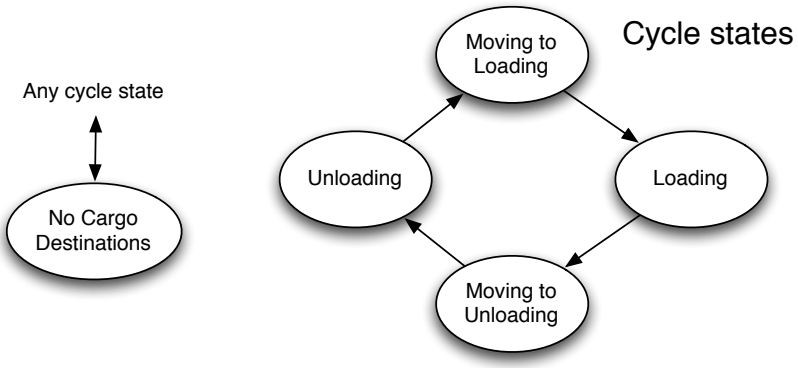
fat interface functions. Throw an Error exception that the action cannot be done.

Tanker Behavior

Tanker also has a set of states that are change in certain ways in synchrony with the Ship state. These states and the possible transitions are shown in the diagram below. A Tanker starts in the No Cargo Destinations state. In this state, it behaves only like a Ship, and will stay in this state regardless of the Ship state. If Cargo destinations are both set, the Tanker will start going through the states labeled as *Cycle* states in the diagram - Loading oil at the loading destination, then Moving to the Unloading destination, then Unloading there, then Moving to the Loading destination, and continuing the cycle indefinitely unless the Ship becomes unable to move or starts sinking, or it is told to stop whatever it is doing, whereupon it returns to the No Cargo Destinations state. When Cargo destinations are both set, the Tanker changes to the most appropriate state - for example, if it is already at the loading destination, it will go into the Loading state and the cycle will start automatically from there. The Tanker will reject many normal Ship commands if it is one of the Cycle states.

Refer to this diagram while reading the following description of how each member function takes the Tanker and/or Ship state into account.

set_destination_position_and_speed, set_course_and_speed. If the tanker state is not No Cargo Destinations, throw an Error. Otherwise, call the Ship::functions of the same name with the same arguments.



Tanker state transition diagram

stop. Do a Ship::stop, and forget the load/unload destinations (set the pointers to zero), and set the tanker state to No Cargo Destinations, and output "now has no cargo destinations".

describe. Output "\nTanker" followed by the Ship description, followed by the cargo amount followed by " no cargo destinations", "loading", "unloading", "moving to loading destination", "moving to unloading destination", depending on the state.

set_load_destination, set_unload_destination. Throw an Error the state is not No Cargo Destinations, Then save the supplied Island pointer, and check if it is the same as the saved value of the other Island pointer, and throw an Error if so. Then output "will load/unload at <island name>". Finally, if both destination pointers are now set, change the state as follows:

1. If the ship state is Docked, and we are at the loading destination, set the state to Loading, and correspondingly if we are at the unloading destination, and do nothing further.
2. If the ship state is not moving, and we can dock at the loading destination, do so, and set the state to Loading, and correspondingly if we can dock at the unloading destination.
3. If the cargo amount is zero, use the Ship member function to start moving to the loading destination position at maximum speed, and do nothing further.
4. Otherwise, similarly start moving to the unloading destination.

update. The initial Tanker state is No Cargo Destinations. First update the Ship state, then do the following in this order:

1. If this Ship cannot move, set the state to No Cargo Destinations, forget the pointers, and output "now has no cargo destinations".
2. If the state is No Cargo Destinations, do nothing further.
3. If the state is Moving to Loading, then if we are no longer moving, and we can dock at the loading destination, do so, and set the state to Loading, and do nothing further.
4. If the state is Moving to Unloading, then if we are no longer moving, and we can dock at the unloading destination, do so, and set the state to Unloading, and do nothing further.
5. If the state is Loading,
   - First, refuel using Ship::refuel. Then compute how much fuel is needed to fill the cargo hold to full capacity. If the amount needed is less than .005,
     - Set the cargo amount to the capacity (to set to "full"), and call Ship::set_destination_position_and_speed function to start moving to the unload destination position at maximun speed, and set the tanker state to Moving to Unloading. and do nothing further.
   - Else more cargo needs to be loaded. Ask the Island for the required amount of fuel to fill up the hold, add the supplied amount to the cargo, output "now has ... of cargo" and do nothing further.
6. If the state is Unloading
   - If the cargo amount == 0.0, call Ship::set_destination_position_and_speed function to start moving to the load destination position at maximun speed, and set the state to Moving to Loading, and do nothing further.
   - Else unload the cargo. Give the cargo to the Island, and set the amount of cargo = 0.0, and do nothing further.

Warship Behavior

Warships also have a state, but it is very simple. A Warship is either Attacking or Not-Attacking, so a diagram is unnecessary. Here is the behavior of Warship functions:

describe. Output the Ship description, then output "Attacking <target name>" if the warship state is Attacking, and nothing otherwise.

attack. If not Afloat, throw an Error - "Cannot attack!" (Warships can use their weapons even if Dead in the Water). If the target is the same as this Warship, throw an Error "may not attack itself". Save the target pointer, set the state to Attacking, and output "will attack <target name>".

stop_attack. If not Attacking, throw an Error. Output "stopping attack", set the state to Not-Attacking, and forget the target pointer (set it to zero).

update. The initial state is Not-Attacking. First update the Ship state. If Not-Attacking, do nothing further. But if Attacking:

1. If this Warship is not Afloat, or the target is not Afloat, stop the attack by calling stop_attack().
2. Else output "is attacking" and do nothing further. (The derived class **update** knows when to actually attack.)

fire_at_target. Output "fires" and call receive_hit on the target with this Warship's firepower.

Cruiser Behavior

update. First update the Warship state. Do nothing further if Not-Attacking.

1. If the target is in range, fire at it by calling the Warship function.
2. Else, output "target is out of range" and stop the attack by calling stop_attack().

describe. Output "\nCruiser" followed by the Warship description.

Controller Behavior - User Input Syntax and Commands

`Controller::run()` first creates and attaches the View to the model, and then starts a command loop. It outputs the current time and prompts for a command, the does the command and prompts again.

Command words and parameters are white-space delimited strings or numbers. As in previous projects, the input is read so that typeahead is allowed, and the rest of the line is read and discarded if the input is erroneous. The command syntax and processing is as follows, done in this order:

1. Read the first word.
2. The first word should be either "quit", the name of a Ship, or a command word.
   - If "quit", then detach and delete the View, output "Done" and return from Controller::run.
   - If it is the name of a Ship, then the next word should be a ship command word. The next items read depend on the command. The information is sent to the ship named in the first command word.
   - If the first word is not the name of a Ship, then it should be a command word for the Model or the View, and the next items read depend on the command. The information is sent to the Model or View, depending on the command.
3. If the first word is neither a Ship name nor a valid command word, then throw Error("Unrecognized command!").

The View commands are:

- **default** - restore the default settings of the map.
- **size** - read a single integer for the size of the map (number of both rows and columns)
- **zoom** - read a double value for the scale of the map (number of nm per cell)
- **pan** - read a pair of double values for the (x, y) origin of the map.
- **show** - tell the View to draw the map - note that the Model should have kept the View up to date by calling plot for every object after updating and adding a new object to the world.

The Model commands are:

- **status** - have all the objects describe themselves
- **go** - update all the objects.
- **create** - read a name for a new ship and check it for validity, throw an Error if the name is already in use. Then read a type name for the new ship, and a pair of doubles for its initial position. Create the new ship using the Ship_factory and add it to the Model. *Note:* No check is made of whether the name of a new ship is the same as one of the commands; the result is undefined.

Controller performs some basic validity checks on the data for ship commands (see the section on Error handling). The Ship commands and basic validity checks are:

- **course** - read a compass heading and a speed (both doubles) for the Ship to set course and speed.
  - basic validity check: 0.0 <= compass heading < 360.0, speed >= 0.0
- **position** - read an (x, y) position and then a speed. (all doubles) for the Ship to set destination position and speed to go to.
  - basic validity check: x, y can have any value, speed >= 0.0
- **destination** - read an Island name and a speed. The position for the Ship to go to is the position of the named Island. Note that ship does not know that its destination position corresponds to an Island - it just goes to that position.
  - basic validity check: Island must exist, speed >= 0.0
- **load_at** - read an Island name.
  - basic validity check: Island must exist
- **unload_at** - read an Island name
  - basic validity check: Island must exist
- **dock_at** - read an Island name
  - basic validity check: Island must exist
- **attack** - read a Ship name
  - basic validity check: Ship must exist
- **refuel** - no additional data required
- **stop** - no additional data required
- **stop_attack** - no additional data required

Model Behavior

Model maintains a integer time value that is incremented when the objects are updated. Model must have the following three containers to support its look-up and describe/update functions.

- A container of Sim_object * -- all Sim_objects created are added to this container.

- A container of Island * -- all Island objects created are added to this container and the Sim_objects container.
- A container of Ship * -- all Ship objects created are added to this container and the Sim_objects container.

These three containers must be kept in alphabetical order by the name of the object. Your choice on the container types.

When Model is constructed, it creates an initial set of Islands and Ships (with the Ship factory) and adds them to its containers. See the supplied files for the order of creation of the specific objects. Observe the order of constructor messages. When an object is removed, it must be removed from both containers holding its pointer. When Model is destroyed, it must delete any objects remaining in the Sim_object * container, doing so in alphabetical order by name.

attach. When a View is attached to the Model, View::update_location should be called for each current object.

add_ship, add_island. When a Ship or Island is added to Model, if there is a View, View::update_location should be called for it.

remove_ship. When a Ship is removed from the Model, if there is a View, View::update_remove should be called for it.

update. Add one to the time, then update all the Sim_objects; the updates should happen in alphabetical order by name. After all objects are updated, if there is a View, call View::update_location for Ship (see discussion below). Then collect all of the Ships that are in the On the Bottom state, and for each one, call View::update_remove and remove it from the containers, and delete it. Note that the destructor messages should appear; observe the order of appearance of the messages. The deleted object should no longer appear in any of the program output, and any Ships that were referring to them should have ceased to refer to them when they became no longer Afloat - your program logic should ensure that there are no dangling pointers to deleted Ships.

View Specifics

View generates a map-like display of where the objects are in the plane. See the example output. The display consists of a square matrix of cells, with each cell containing two characters. The size of the matrix is initially 25 X 25 cells, 25 rows and 25 columns. The constructor sets the matrix to the "empty" pattern, which consists of a period (`'.'`) in the first (leftmost) character and a space in the second (rightmost) character of each cell. When output, this produces a "grid" effect.

Each cell represents a range of (x, y) coordinates; the size of each cell is the scale. For example, the initial setting is that each cell represents a 2 X 2 square of arbitrary units. In the output, larger values of y are at the top, and larger values of x are to the right. The lower left-hand corner of the display is called the origin; the initial setting is that the origin is at coordinates (-10, -10). There are member functions that allow the size, scale, and origin of the display to be changed, so that any part of the plane can be covered in whatever detail is desired. View throws Errors if one of the functions receives invalid values. These checks are as follows:

- The map size must be <= 30 and > 6.
- The scale must be > 0.0
- The origin can be any point.

The default settings for the matrix are size: 25 X 25, scale = 2.0, origin = (-10, -10). These settings can be restored with the set_defaults function.

To produce the display, each object to be plotted must have been used in at least one call to `update_location`, which takes the object name and location as arguments. This information is simply remembered. If the `update_remove` function is called, the name and location is "forgotten" and the object will not be plotted. Then to output the map, call the `draw` function. The position of an object is plotted by showing the first two letters of the object's name in a cell, where the cell row and column is produced by translating and scaling the object's position according to the origin and scale of the matrix, and then converting the resulting position to integer subscripts. The function `get_subscripts` is provided to ensure uniformity in this computation. If two objects are plotted in the same cell, an asterisk (`'*'`) and a space are placed in the cell.

The draw function outputs the map matrix to produce the display like that shown in the sample output. The size, scale, and origin are printed first. If the subscripts for a particular object lie outside the matrix, a message "<object name> outside the map" is printed next. Multiple objects outside the map are listed separated by a comma and a space. This is not an error; it simply informs the user that at the current size, origin, and scale, not everything can be seen. Then each row and column for the current size of the map is output.

If the map size, scale, or origin is changed, and the `draw` function is then called, it has to output the map with the new size, scale, or origin with the all of the objects correctly located. This can be done because View "remembers" the objects' names and locations given to it in calls to `update_location`.

The x and y axes are labeled with values for every third column and row. You must use the output stream formatting facilities (Output Formatting handout) to save the format settings, set them for neat output of the axis labels on the grid, and then restore them to their original settings. *You must use the stream output formatting functions and manipulators for this purpose. Do not try to "fake" it with your own string-processing code - it will be much harder, and much less educational.* Allow four characters for each numeric value of the axis labels, with no decimal points or places to the right of the decimal point. The axis labels will be out of alignment and rounded off if their values can not be represented properly in this format. This distortion is acceptable in the name of simplicity for this project.

A convenient way to create and output the map matrix is to use the remembered information to populate a three-dimensional array, and then output each row of the array. Note that the array does not have to persist between calls to draw() - the names and locations of objects are what must be remembered - the array might only be local to draw(). If you have not worked with a three-dimensional arrays before, I recommend implementing View using a built-in array of char with sizes 30 X 30 X 2. While not the most modern approach, it will be good practice and applicable to C programming as well. If you have worked with multidimensional built-in arrays before, try implementing View using a `vector<vector< vector <char> > >`, or `vector<vector<string> >`, which once created, can be used syntactically with subscripts just like a built-in array. Note that vector has a constructor that lets you preset the size of the vector, and a resize function, both of which could be useful.

Model-View-Controller Interaction

In the Model-View-Controller pattern, Controller normally has the responsibilities of creating the Views and attaching them to to Model, and then detaching them and destroying them when they are no longer needed. In this introductory simplified case, at the start of the run function, Controller will create one View object with new and attach it to the Model, which keeps a pointer to only one View. When Controller terminates, it detaches the View from the model, and then deletes it.

The sequence diagram below shows the basic pattern of interaction between the human user, the Model, View, and Controller objects, and a specific Ship object that gets created in response to a user command.

The View object receives the information about which objects to plot from the Model. Model updates the View by calling `update_location` with the name and location of Sim_objects. Model updates View about *all* of the Sim_objects currently in existence when `Model::attach()` is called. When an Island or Ship is added, Model gives View an `update_location` about that one new object. But subsequently, after `Model::update()` has updated the Sim_objects, Model calls `update_location` only for the objects that may have moved, namely the Ships. If a Ship has been removed from the Model, Model calls the View `update_remove` function to remove that object from the plot. Since Islands don't disappear at this time, they are never removed from the plot.

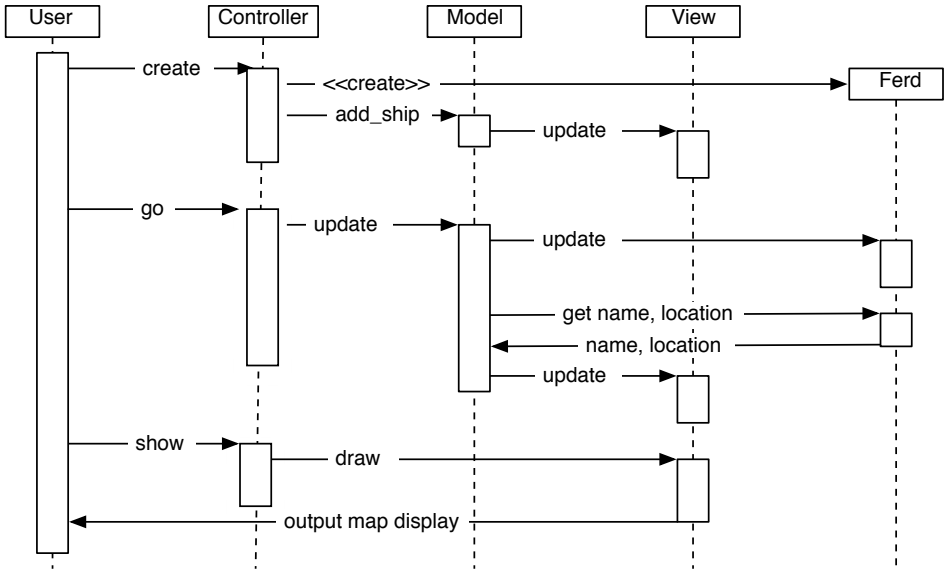
View saves the update information it receives, namely the name and position of each object currently in the plot. Then when the user asks to see the map, Controller calls the draw function to output the map. Because View "remembers" which objects it needs to include in the map, it can output the display at any time; the display should always be consistent with the state of the Model, but at the same time, because it only remembers names and locations, it is decoupled from the Model and the Sim_object classes. Also because of its memory for the plotted information, View can output the display correctly after Controller has told it to change the size, scale, or origin of the map without requiring a fresh update from the model.

Thus every time the state of the world changes, Model tells View about the changes. In this simple implementation, on each Model update, Model tells View only about the objects that *could* change positions (currently just the Ships - no flying Islands). With additional complications, it could update View with only the Ships that *actually* changed positions, but we'll not get this complicated in the project. Complex and efficient versions of this design pattern are certainly possible, and are very important in GUI implementations.

Constructor and Destructor Messages

It is important to understand the order in which construction and destruction is done in a class hierarchy. For this reason, each of the classes involved in the Sim_object family must print a message in its constructor outputting the name of the object, and another message in its destructor doing the same. The Model, View, and Controller objects must do likewise, however, they have no individual names. Track_base already includes the desired message. See the project starter files for the text of these and the other messages.

Project 4 Sequence Diagram
Showing Basic Model-View-Controller Interaction



The user creates a ship named "Ferd", and issues a go command. Controller tells Model to update; Model updates each Sim_object (only Ferd is shown), and then gets the name and location of each Sim_object, and updates View with the information. When the user asks, Controller tells View to draw, producing the map output.

Error handling

Because this is a highly interactive program, it must be prepared for the many errors that human users will make. The program must deal with all of these in a simple and uniform way by using exceptions. At the point where an error is detected, an Error object is created to contain the error message and then is thrown. The command loop in Controller contains a try-block followed by a catch for Error objects. The catch block outputs the error message, skips the rest of the cin input line still waiting in the stream, and then the program prompts for a new command. The exact order in which the stream is read, and errors checked for is important: the goal is for this to be as uniform and consistent as possible.

The principles for the error checking is that errors are detected and the exception thrown in the component and function that has the information and the responsibility for that information. The general pattern for the error checking of the user input is as follows: Do the basic validity checks on each data item immediately after reading it, and before reading the next data item. If a data item is found to be invalid according to the basic validity check, the rest of the input line is skipped. Once all of the input data items for a command are collected and pass the basic validity checks, Controller will send the information to the relevant component.

- Controller is responsible for the following basic validity checks of input data.
  - The input will be read one data item at a time and checked for basic validity one data item at a time until the complete command has been read. A data item is a single string or number.
  - See the input syntax for specifics, but if a command is not a recognized command word, an error is thrown.
  - If a number is expected, but cannot be read (a stream failure), the error is thrown immediately.
  - If a number was successfully read, but is an illegal or impossible value for the parameter according to the basic validity checks listed for Controller commands, an error is thrown immediately.
  - If a string is supposed to be a Ship or Island name, the associated pointer must be obtained from the Model, which thus checks for validity. Names must be unique(across both Ships and Islands), and except for new Ships, must belong to objects that already exist.
- Model
  - If asked to supply the pointer for a Ship or Island, throws an exception if no Ship or Island has that name. Only the Model knows what Islands or Ships exist, so that is where the error is detected and thrown. Controller is not supposed to be responsible for this. Two special cases: Controller will ask Model whether the first command word is a Ship name; if not it will see if it is a valid non-ship-command word before throwing an unrecognized command exception. If a new ship is to be created, Controller will ask Model if the name is already present, and Controller will throw the exception with a special message if it is.
- View
  - Once the information for a complete View command has been obtained, the command can be issued to the View. If the command contains data that is illegal or can't be executed, the relevant View function throws the exception. Only the View knows its state and possibilities, so that must be where the error is detected and thrown. Controller is not supposed to be responsible for this.
- Ship classes
  - Once the information for a complete ship command has been obtained, and passes the basic validity checks, the command is issued to the ship. If the command is illegal for that type of ship, or can't be executed because of the state of the ship, the relevant function in the ship's code throws the exception. Only the ship knows its actual type or state, so that must be where these errors are detected and thrown. Neither Model, Controller, or View is responsible for trying to figure out what a Ship is capable of.
- Ship_factory special case
  - The create command is a special case. Controller is responsible for checking that the new ship name is valid (using a Model service). It is also responsible for getting the position of the new ship, which involves the basic validity check for two doubles. However, only Ship_factory knows what the valid ship type names are. So in a departure from the general pattern, the ship name is read and checked, then the type name is read and held, then the position is read (and checked for doubles one at a time). Then the name, type name, and position are sent to Ship_factory, which will then check the type name.

General Requirements

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

- The program must be coded only in Standard C++.
- You must follow the recommendations presented in this course for using the std namespace and header file contents.
- Your use of the Standard Library should be straightforward and idiomatic, along the lines presented in the books, lectures, handouts, and posted examples.
- You should use the algorithms, adapters, and binders at least where they work easily and simply. More specifically, if you can write a one-liner using the STL (such as for_each with mem_fun and say bind2nd) or a correspondingly simpleuse of TR1::bind, then you should. Otherwise, you can take your choice between custom function objects, elaborate uses of TR1, or explicit loops.
- You *must* use a map of strings to Controller member function pointers to go from input commands to specific command functions - similar to the previous project, except Controller member functions are being called.
- Only C++ style I/O and memory allocation is allowed - you may not use any C I/O or memory functions in this project.
- The program should be as "bullet proof" as possible and should run without crashing regardless gibberish typed in by the user.
- You may not use any global variables anywhere for any reason. The Standard Library global objects cout and cin are the only exception.

- Each file should only #include the header files that are actually necessary. This especially applies to the Sim_object family headers. An unnecessary #include involving them produces unnecessary coupling and is a major error!

Suggestions

While the specifications are complex, this is due mainly to the need to spell out exactly what update sequences and computations need to be performed; in fact, relatively little code has to be written; the hardest code is in the Ship and View classes.

This project is too big and complex to implement all at once, but it greatly rewards building and testing it a bit at a time. You can do unit- and component-level testing of each class with a simple testing driver that creates an object and calls each member function with various inputs and then interrogates the state of the object with other member functions to see if the object is behaving correctly.

An easy start would be to implement and test the Island class this way. Then try implementing Ship as a (temporarily) concrete class and test its movement and sinking behaviors. You could then try building and testing the Warship and Cruiser classes.

At some point, you will likely find it easier to start building the Model class to make it easier to set up your other classes to interact with each other, both to test them as components, and to begin some integration testing.

Create a testing drive whose main() function creates a Model object and let it create a couple of Islands. Have your code tell Model to describe the Islands, then update them, then describe them again. Those with fuel production capability should show the right change. Then have Model create one of your concrete Ships, and then devise tests that tell it to move, update it and describe it, etc. Despite its boring behavior, the Tanker class is actually somewhat subtle, so save it for later - in fact, if your Model and Controller are working well, and you have component-tested Tanker, you should be able to "plug in" the Tanker class and have it "play" immediately! Being able to plug and play new derived-class components is a sign of a good OO framework, and can be really fun!

The major choice point is when you implement View. View is basically simple but nit-picky, especially if you haven't done output formatting before. Read the handout, if so! View can make it easier to tell what you program is doing, but the textual output of describe, and the blow-by-blow provided by step-by-step updates, is the best way to ensure that the Ships are behaving exactly like they should.

Controller can be made neat and easy to work with if you use the required map container that translates strings to pointer-to-member functions to call Controller command functions. This is similar to what you did in project 3, but instead of pointers to regular functions, you store pointers to Controller member functions. Easy, and good practice!

Project Evaluation

The autograder will test your program's output separately using only update output, only update and describe output, and only update and View output. This will help isolate any issues having to do with details of output. If your program fails to match the posted samples due to occasional differences in numeric values or map positions, please let me know promptly in case it is due to some floating-point problems; likewise if it matches the posted samples, but fails to match the corresponding sample tests on the grading machine.

The autograder will also mix and match some of my components and your components, so adherence to the public and protected interfaces is critical. Also, make sure that the output message strings provided are produced by the specified components and functions only - other component testing assumes this division of output production responsibility. Only autograder evaluation is planned for this project.