Project 6
Extending and Enhancing the Ships Project

Final Deadline for all deliverables: 5:00 PM, Tuesday, Apr. 18, 2017.
5% bonus: Submit all materials by 5:00 PM Monday Apr. 17.

Notice:

Corrections and clarifications posted on the course web site become part of the specifications for this project. You should check the this page for the project frequently while you are working on it. Check also the FAQs for the project. At a minimum, check the project web pages at the start of every work session.

Overview

The purpose of this project is to provide further practice with design using the concepts presented in the course. Your task is to choose some "features" to be added to your Project 5 solution, work out a detailed specification of the features, and then design, code, and test their implementation.

You are expected to use the design and programming concepts and approaches presented in this course. The concepts presented in the course can be directly applied to produce elegant results in this project. Since good design is much harder than good coding, you need to start thinking about the design of this project days before you start coding - procrastinate on the coding if you must, but procrastinating on the design will probably be fatal!

The project will not be autograded for output correctness because you control what the correct output is supposed to be. I will also not test your components because you control what the components and their interfaces are. However, you must submit a working version of the program and demonstration samples that allow me to use your new features and verify that they work. In addition, you have to supply some hard-copy documents that specify your additional features and explain your design for them. These *deliverables* are described below.

Each feature is only described at the level of an "idea" - like something the marketing people might come up with. You have to specify the feature - work out the details of the feature and how it should behave - as well as how to design and code it.

If working by yourself, you have to choose two features: One is a *Simple Extension* to the existing Project 5 structure. The second is a *Major Enhancement*. If you are working alone, you have to choose one of each type from the following lists. See below for information about forming a team, and what a options a team can choose. It is a good idea to consider all of these options, both Simple and Major, before making your final choices.

Simple Extensions

These Simple Extensions are intended to illustrate the "plug and play" or "add features by adding code, not changing code" aspect of OOP. Therefore, if your Simple Extension idea requires changing a significant amount of existing code, or much change in the public interface of existing classes, it is a poor choice. Likewise, it should not be necessary to refactor or rearrange the classes that you should have had in Project 5. The Cruise_ship and Local view classes added to Project 4 are good examples of Simple Extensions.

Important: You should complete the Simple Extension(s) first, and in fact it must be possible to implement S1 or S2 so that these are in place before you implement your Major Enhancement. That said, there are some compatibility issues noted for each Simple Extension, so you should consider all of the Major Enhancements before you make your final decisions about the Simple Extensions.

S1. Another kind of ship. Explore the generality of the Project 4/5 framework by designing and implementing a new class of ships. Take advantage of the Project 5 framework; any modifications to it should be in the direction of making similar future additions easier. This new ship type must have the following characteristics:

- It may not be simply a clone or near-clone of the existing three classes - it must behave differently in a significant way. For example, a new class that is nothing more than a more powerful Cruiser will not do.
- It must display "automatic" behavior at least as much as Tanker, Cruise_ship, or Torpedo_boat, and definitely more than Cruiser. If you choose M3 (below), first implement S1 with a single response to being attacked.

Any necessary modifications to the existing framework and classes should be minor and in the direction of making future ship type additions easier, similar to how we added a Model service to enable Cruise_ship and Torpedo_boat.

S2. Another kind of view. This is another kind of view that shows information about the state of the simulated world, either for all or most objects (like Map view and Sailing data view), or for a particular object in relation to other objects (like Local view). Your new kind of view should show information about object states.

The new view type can show additional kinds of information, or an interestingly different combination of the same information, as in the Project 5 views. For example, it could be similar to the "plotting table" commonly used in navigation or the "strip map" used in many GPS device display, such as: ownship is always in the center of the display, the current course is always at the top, and objects are only shown if they are fairly close to ownship. Your design not only must be compatible with the existing views, but will also allow for easily creating new views that have mixtures of the Project 5 view information and any new information types.

Exclusions: If you implement M1 or M2, S2 may *not* be another kind of view that shows player states or group membership - these are too big a departure from the existing view types, and so are not a Simple Extension to the existing view architecture. Such views also have the problem that you cannot implement them before the Major Enhancement, which is not how the Simple Extensions are supposed to work in this project.

Major Enhancements

The Major Enhancements require designing new classes and/or designing a rearrangement of existing class structures. This re-architecting of the project is most easily done if the Simple Extensions are already in place. Each Major Enhancement has a statement of Design Goals which describes what a quality design will accomplish. Your design must meet these goals.

M1. Multiplayer game with computer-controlled players

Project 5 was in the form of a simulation, in which the user could issue commands to any ship, and had available a single map view of the entire world. This new feature is to change the simulation into a multiplayer simulation game in which any number of players would take turns; the "tick" of the simulated time clock happens only after all players have said "go."

Each player has their own set of Ship objects that only he or she can control, and islands on which they have a base, and her or his own set of views; a human player can open whichever views they want, and only that player can close those views; they continue to exist across turns, retaining their settings, until the owning player closes them. However, a player’s views can be displayed with **show** only during that player’s turn (we'll assume that players do not try to look at the display while some other player is taking their turn).

In addition, it must be possible to have any number of computer-controller players, although for purposes of this project, they are "stupid" - developing the AI for good computer-controlled players is well beyond the scope of this project. So the computer-controlled players need to do something very simple, such as dispatching a ship to somewhere, attacking another player's ships, or starting Tankers moving oil between its Islands. The point here is just to demonstrate that you have made a place where smart computer-controlled player functionality can be "dropped in" at some point in the future. Note that for design purposes, you can assume that a computer-controlled player can directly control its Ships, and the code should only perform actions that are legal (even if stupid). Of course, computer-controlled players do not show any views - they can’t see them.

Any number of humans can play and have any number of objects or views. The number of players of both types is specified when a game is started, but a player can leave the game during their turn. You have to decide and specify when a player is required to leave the game (e.g. no more Ships left afloat), what happens to a player and its stuff when the player leaves the game, and when the game is over (e.g. only one human player left).

There must be a place in the project where additional game-like resource features could be added; for example, other kinds of resources besides oil, or perhaps certain resources are required before new ships can be built. You must be explicit on where such resource information would be held and used, but actually implementing resource features is optional. However, take care not to get carried away: these additional features should not be designed and implemented unless and until the basic multiplayer features are in place and well-designed.

Design goals: It must be possible to add additional Ship, View, or Island types with no change to the multiple-player code. Even if you don’t add additional game-like resource features, the framework must be in place to add them with little or no change to the rest of the code that is not immediately involved. For example, if we decide that a player has to have a certain amount of oil in order to build a ship, it should be possible to add this constraint in a way that only affects the code relevant to a player building a ship. Similarly, it should be possible to make the computer-controlled players smarter with little or no changes to other code.

M2. Groups of Ships

Provide ways for the user to put ships into groups and control the groups as a whole, in addition to the existing individual ship capability. You will recognize this feature from games such as Warcraft: You can form units into groups at any time and move or command the group as a whole, and any unit can be removed from the group and controlled individually, or groups disbanded, at any time. Unlike Warcraft/Starcraft (at least the versions I've seen), any number, kind, or mix of units, including groups, can be included in a group. In addition:

- You have to decide what a legal group structure consists of within this requirement, for example, whether a unit can be in more than one group - and design and implement how to enforce it.
- You have to decide how ships in a group should move, be displayed, attack, etc., especially if they are of different types, and come up with a good design and good implementation code.

Design Goal. There should not be any limits, either current or future, on which kinds of ships can be formed into groups and controlled as a group. In other words, if a new ship class is added in the future, it must be possible to treat it as a member of a group with no changes to the group control code.

M3. Selectively Responsive Ships

Currently some types of ships can attack other ships, but a ship's response to an attack is always the same. For example, Cruise_ships and Tankers just get sunk, Cruisers counter-attack, and Torpedo_boats run away. We want to have the ships respond in more interesting ways that depend on the type of the attacker. For example, suppose we have two warships, and one attacks the other. Each ship behaves differently depending on the type of its opponent. For example, if a Torpedo_boat is attacked by a Cruiser, it attempts to run away, but if attacked by another Torpedo_boat, it counter-attacks. Likewise, if a Cruiser is attacked by another Cruiser, it could counter-attack as it currently does, but if attacked by a Torpedo_boat, it could fire back and start chasing it. (Feel free to do this differently - it is just an example).

Even though Tanker and Cruise_ship themselves cannot attack, they need to respond differently depending on the type of Ship attacking them. You can increase their resistance so that they have a chance to respond instead of just sinking quickly. In addition, you can modify the resistance, firepower, maximum speed, etc. of any of the ships if helpful to make a good demonstration of this feature.

It must be possible for all ship types to respond selectively, even if you choose in some cases to implement responses that are the same for some of the ship combinations (e.g. running away). If you chose S1, you must include it as part of M3, adding additional responses to your S1 version.

Important: The basic approach must allow for any *arbitrary* type or pattern of interaction; for example, a fixed pattern based on a simple classification of ships such Warship vs Non-Warship, or simply based on relative firepower, is not acceptable. The design should support any kind of responsive behavior that can be reasonably programmed.

Design goal: The approach taken to provide this feature must easily extend to additional types of ships in a straightforward way and must be able to represent any arbitrary interaction. If we add an additional type of ship, then it must be possible to easily implement its interactions with the existing types of ships. This needs to be done in a way that gives a good tradeoff in the various issues involved, and corresponds well to the guidelines of good OO design.

M4. Save/Restore

Note: This option can be used only by a team of three in combination with another Major Enhancement. Give the user the ability to save the state of the program, including any open views and their state, to a specified file at any time. At any time, the user can restore the program state from a specified file - the result is a simulation (or game, if M1) and views that are in the exact same states as they were at the time of the save. This must be done in such a way that if the program is expanded, e.g. by adding another type of ship or view, it will be easy to include the new information in the save/restore system with little or no modification to existing code.

What you have to achieve in the Major Enhancements

The major criterion for a good design is whether the program can be easily extended, the hallmark of good OO design. This includes not just the clarity and simplicity of the code, but also whether additional classes or subclasses of the various types can be easily added to the extend the new capabilities of the program, and in a way that requires little or no modification of existing code — "adding functionality by adding code, not by modifying code."

Each feature contains a concise statement of the design goal, which involves developing a general capability of some sort that will support future additions or modifications along the same lines. Your problem in the project is to (1) achieve this general capability with an extensible design, and (2) demonstrate that it works with some specific example implementations.

Two pitfalls to avoid: (1) Implementing a design that does your examples in the minimal possible way without providing a clear extension pathway - in other words, failing the design goal - this misses the point of the project. (2) Running amok with example implementations, such as a dozen new kinds of ships for M3 — this will be a waste of time and it will distract from creating a good design. A lot of examples won't make up for a defective design, no matter how clever the examples are.

So keep the priorities clear: The primary goal of the major enhancement is to *provide a general capability for extensions of a certain kind*. The specific implementations are secondary, and are required only to demonstrate the scope and power of your general capability. In other words, many clever specific implementations in a poorly designed framework will be worth very little, while a few well-chosen implementations that demonstrate the scope and power of a well-designed framework will be considered an excellent result. Please ask if this is not clear.

Other Rules

Programming requirements

This project is to be programmed in Standard C++14, and take full and appropriate advantage of the Standard Library facilities covered in the course. Your program must follow the course C++ Coding Standards and the guidelines and concepts for good software design and coding presented in this course. Using additional features of Standard C++14 beyond those covered this semester is permitted, but is definitely not essential for a good design, and can result in convoluted or over-engineered code (*Do simple things in a simple way!*). So take care not to get distracted.

What You Can Change

The required design elements and components of Project 5 must be present in this project, because, as described for Project 5, this is the second part of a two-part project. To be clear, these are the Model Singleton, pervasive use of smart pointers, the MVC organization, the three kinds of views, and the three kinds of ships.

As long as these design elements are present, you are free to modify the Project 4 and 5 classes as you choose, but keep in mind that this project is supposed to be based on Projects 4 and 5, so the more of that code you can re-use in this project, the better off you will be. Where applicable, the project should be fairly close in its behavior to Project 5 — this is not a new project, but an extension of the previous ones. All previous features should continue to be present, although details might be different and they might be implemented differently. Thus a complete rewrite is not called for, and is almost certainly not needed, but you can make any changes that will help you achieve a good design in this project.

What You Must Change

Now that the tyranny of autograding is over, take your Views.h, .cpp Project 5 file apart into the usual configuration of one file pair per class. Thus you'll be submitting separate .h, .cpp file pairs for the different Views instead of just one pair. Similarly, separate the classes in the Warships.h, .cpp Project 5 files into one file pair per class. Do not submit Views.h, .cpp; do not submit Warships.h, .cpp.

Teams

Object Oriented Programming can work well with development teams. The team members first come to agreement on the responsibilities, collaborations, and public interfaces of the classes in the design, then divide the classes up between the team members, and then each team member develops the private implementation for his or her classes. If done properly, the separate classes will plug-and-play together, and changes and refinements to the design can be easily worked out and implemented.

If you want to try this out, you may form a team of up to three persons for this project.

Pitfalls. Teams can be great, but there are some problems to consider:

- Additional features are required for a team project, and team coordination consumes time. Thus working on a team will probably be more total work than working individually.
- The result will be better than individual work only if the team makes a point of constructive criticism of each other's work. It often seems that team members support each other in doing poor-quality work - you don't want to tell your friend that his idea or code stinks. So it would help if members are willing to reveal to each other how well they have done on the previous code quality evaluations, so that their individual strengths and weaknesses are known; I have seen situations in which the weakest programmer apparently dominated the final result. Instead, all team members need to keep the project on track to meet the requirements and meet them well. Ask each other, "What would Prof. Kieras think of this?"
- Another extreme: the team talks themselves into a very ambitious and well-designed project that maxes out the grading scheme but involves more work than necessary. I can't reward the extra work, and it can be so much that it interferes with the team's other courses. Come and meet with me about your ideas if this might be a problem.

Your team has to start with a Project 5 implementation. You can choose a single solution authored by one of the team members, or you can combine pieces from more than one member of the team. You should review and compare your solutions, pick the best or best parts, and fix any problems you identify, because the Project 5 implementation will be human-evaluated as part of Project 6, and all team members will get the same score.

A team must supply an additional document on how the design, implementation, and documentation work was handled as a team. It is expected that each member will make substantially equal contributions to the project, and all will write code - no "documentation specialists!"

Team options:

- An individual person must do one of (S1, S2) and one of (M1, M2, M3).
- A team of two people must do both of (S1, S2) and one of (M1, M2, M3).
- A team of three people must do both of (S1, S2) and two of (M1, M2, M3, M4).
- Teams of more than three are not permitted.

Choose one member of the team to be the "lead" for submission purposes. The code will be submitted to the autograder by this team member.

Special communication requirements for Teams. Team members must keep each other fully informed about anything that affects the team. Thus any and all email communication with the teaching staff (Prof and TA) needs to cc all members of the team. If you meet with teaching staff during office hours, all members of the team should be present if possible. If a team member encounters a need for an extension because of health or other problems, it is essential to follow the Syllabus & Policies rules about communications in this case.

Deliverables

Autograder deliverables

We will be using the autograder simply as a way to send in your code and do a check compile of it. The result will be only 0 or 1 points for a failed compile or successful compile, respectively. These points will not be counted in the project score. Here is what you must deliver and how you must deliver it:

1. If you are part of a team, the lead team member must submit the autograder deliverables. To avoid confusion, other members should not submit any autograder deliverables.

2. You must submit your source code and a makefile. The makefile must be named `Makefile` and the command `make` with this file *must build an executable named* `p6exe`.

Your code must compile and run without error in gcc 5.1.0, using the submitted makefile, and C++14 options, and must be complete as submitted - I will not supply any files of my own. You should do a check compile and run in the CAEN gcc 5.1.0 environment before finalizing your submission.

3. Along with your source code and makefile, you must submit a set of interaction demonstrations similar to the samples provided with the previous projects that demonstrate your new features. Each demo consists of an input and an output text file and must be suitable for I/O redirection, along the same lines as the sample files that have been provided in the course projects. The files *must be named* `demo1_in.txt`, `demo1_out.txt`, `demo2_in.txt`, `demo2_out.txt`, etc. There must be at least one such pair of files for each feature (S1, S2, M1, etc), with a maximum of 10 pairs for the project. These files should correspond to the annotated hardcopy demonstration console documents, explained below.

4. Your submitted files, both code and demo files, *should be all in the same directory with **no** subdirectories*.

5. If you have correctly set this up, the following command sequence issued in this single directory should execute correctly, and produce no differences when repeated for each set of demo files:

```
make
./p6exe <demo1_in.txt >test1_out.txt
diff demo1_out.txt test1_out.txt
```

Delete any left-over text files from the testing, and then the following submit command will correctly submit your project:

```
submit381 6 Makefile *.h *.cpp *.txt
```

Check the submit381 feedback to verify that you've sent all of the required files.

6. Before finalizing your autograder deliverables, review #1 through #5 above and make sure you get them right. Do not lose lots of credit simply by not submitting these files correctly.

7. After the project deadline, I will run a checking script that runs your p6exe with each demo _in.txt file and diffs the result with the corresponding demo _out.txt file. My goal in running your program with (and without) your demos will be to assess whether your program actually does the things you specified. You will lose credit if there are problems that cause inconvenience or prevent me from compiling and running your code, such as misnamed files, compile errors due to non-standard code, missing files, or run-time errors that interfere with running the program. *I will not attempt to fix your submission in any way whatsoever.* See #6 above.

Hardcopy deliverables

You must submit some hard-copy paper documents in addition to your code. The quality of these documents is more important than the reliability of your code - plan plenty of time to prepare them! They are as follows, but here are the general requirements for the documents:

- You only supply documents for the features (S1, S2, M1, M2, M3, M4) that you chose to implement. Do not include any material about the features you did not implement.
- For each feature you chose to implement, you must supply a *description* document, one or more *demo* documents, and a *design* document. The requirements of the design document are different for simple and major features.
- A team must supply a team activity document.
- Demo documents can be in any convenient font and layout; the description, design, and team activity documents must be written single-space, 12 pt Times or similar font, 0.5 - 1.0" margins, preferably printed on both sides of the paper.
- If a demo, description, design, or team activity document is more than one sheet of paper in length, *the pages in that document must be stapled together.*

1. Description (How it works/how to use it) document for each feature. This document identifies the Simple Extension or Major Enhancement that you chose (that is, S1, S2, M1, M2, etc.) and describes the specific details of how the new feature *behaves* and *how to use it* - it corresponds to what might be in the user manual for the program, and thus it is essentially your specifications for the new features. It does ***not*** describe the design or implementation of the feature! The Project 5 specifications for Cruise_ship, Torpedo_boat, and the new Views are a good example of the level of detail and approach you should write for the description document - notice how they basically describe *behavior*, *not* design, and *not* implementation.

I will assume that everything specified in Project 4 and 5 still applies, so you only need to describe Project 4 and 5 features and commands if you have changed them in order to implement this Project. Length: about 1 page/feature.

2. A hard-copy annotated console demo document(s) similar to the console samples in previous projects, showing the input and output of your program for the demo in/out files that you have supplied. The demo should be annotated on the hard copy with explanations of what is happening - the annotations should be written by hand on the hard-copy. The document should have a name written on it that corresponds to which of the "demoN_in/out.txt" redirection text file pairs it corresponds to. The submitted demo files, described as Autograder Deliverable #3 above, are the corresponding input/output files that I can use for redirection. By running your program using the demo files, and examining its behavior and this hardcopy document, I should be able to see your new program features in action, and understand how they behave. You can have one demo document per feature, or combine them as convenient. Length: as needed.

- Note that I will also "play" with your program some — it should not hang, get confused, or crash if I depart from your samples.
- To be acceptable, this document must show *both the input and output* with your annotations. It's a good idea to figure out how to capture the console transcript early in your work; details differ depending on the platform. Don't take the risk of this being a last-minute show-stopper.

3. Design document for Simple Extension(s). The behavior of the Simple Extension was described in your feature description document, and should not be repeated in this document. This document simply explains how the Simple Extension fits into the rest of the project design; it should be fairly simple if you chose a good "plug and play"design approach, and no or few modifications to existing classes. Length: About 1 page or less per feature.

4. Design document for Major Enhancement(s). Each Major Enhancement that you designed and implemented in this project requires a document that describes the design. The purpose of this document is that after reading it, I should be able to understand your code much more easily, and understand why you organized it the way you did. These documents can assume the Project 4 design - they do not have to describe the Project 4 design except where it was changed, or where it is helpful in presenting the Project 5/6 design.

The design document for each Major Enhancement that you designed and implemented must include:

- *A design presentation* that explains the design of the Major Enhancement, referring to the diagrams (described below), and using the terminology from the course materials for any patterns, idioms, or concepts that play a role. OOP programmers use this vocabulary to improve communication, and you should too. For example, if you are using the Abstract Factory pattern, use the name "Abstract Factory". A key writing point: the text should refer to the diagrams and tell the reader what he/she should be understanding from the diagrams. Length: about 2-3 pages.
- *An extensability statement* that explains how the design goal of easy extension would be met for the Major Enhancement. For example, in Major Enhancement M3, explain what would have to be done to add another kind of ship and make it respond differently to the existing ship types, and vice-versa, and how you have made it easy to do. Length: About one page or less.
- *A UML class diagram* showing how the Project 5 classes relate to each other, along with any new classes in your Project 6 design. The Standard Library smart pointer, container classes, and similar "utility"classes should not be included. Also, you can follow the example of the Project 4 class diagram and show only the key members of each class - those that are important for understanding the design. This diagram can be hand-drawn as long as it is clear. (Don't waste time learning a drawing tool just for this project). Refer to the UML handout and the lecture notes and follow the format and examples in the Project documents - for example, inheritance is shown with vertical, not horizontal, connections. Be sure the diagram is correct - for example, check that there are no missing connections between classes. If your team implemented more than one Major Enhancement, and they can all be presented in one UML diagram, then only one diagram needs to be supplied and the design document for each Major Enhancement can refer to it.
- *A UML sequence diagram* that illustrates an *informative* interaction between objects in the Major Enhancement. This interaction should be something basic to the feature design and that helps me understand how your design works. This can be hand-drawn as long as it is clear. Refer to the UML handout and follow the format. Note that this diagram shows the interaction between objects, not classes; make sure that yours does likewise.

Important: Reading the design document should make it easy for me to understand the structure and organization of your code. If the documents are incomprehensible or incomplete, not only will you lose credit for them, but I will not take the extra time to figure out your design from the code, and so will downrate its design quality as well.

5. Team activity document. If a team, you must supply a document that lists the team members, describes whose Project 5 code was used (or what parts from which team members were used), describes how you arrived at the project design as a team, and which team member was responsible for what work in the project. Length: 1 page.

Hardcopy Deliverable Examples

One person does S1 and M1; six or more separate documents are required:

- S1 description doc - if more than one sheet in length, staple the pages together.
- S1 design doc - if more than one sheet in length, staple the pages together.
- S1 demo document(s) - at least one is required, staple the pages for each demo document together.
- M1 description doc - if more than one sheet in length, staple the pages together.
- M1 design doc - staple the pages together.
- M1 demo document(s) - at least one is required, staple the pages for each demo document together.

A team of three persons does S1, S2, M1, M2; twelve or more separate documents are required:

- S1 description doc - if more than one sheet in length, staple the pages together.
- S1 design doc - if more than one sheet in length, staple the pages together.
- S1 demo document(s) - at least one is required, staple the pages for each demo document together.
- S2 description doc - if more than one sheet in length, staple the pages together.
- S2 design doc - if more than one sheet in length, staple the pages together.
- S2 demo document(s) - at least one is required, staple the pages for each demo document together.
- M1 description doc - if more than one sheet in length, staple the pages together.
- M1 design doc - staple the pages together.
- M1 demo document(s) - at least one is required, staple the pages for each demo document together..
- M2 description doc - if more than one sheet in length, staple the pages together.
- M2 design doc - staple the pages together.
- M2 demo document(s) - at least one is required, staple the pages for each demo document together.

Submission Rules

Because paper documents must be turned in, the deadline for each day (and early submission bonuses) is at 5:00 PM instead of midnight. To give me a chance to get started on evaluating projects as they are turned in, your project will be considered as completely submitted as of 5:00 PM on the day when you deliver the hard-copy documents; that is, after 5:00 PM on that day, I will assume that I can run and print out your code and start evaluating it.

So don't turn in the hard-copy deliverables until you have submitted your final code.

Warning: The announced deadlines are HARD deadlines. *Even one minute late is not acceptable*. If you run out of time, turn in an incomplete project instead of trying to turn it in after the deadline.

Because of past traumatic and miserable experiences with handling large numbers of paper documents for many projects, you *must* follow these rules for the hardcopy deliverables:

- Your name or uniqname must appear on each paper document (if a team project, the names of all team members should appear on each document).
- I will not fuss with a zillion loose sheets of paper when I have dozens of projects to evaluate. Stapling multi-page documents as described above is mandatory.
- The paper documents must be enclosed in a 10" X 13" clasp-type envelope (see the picture). Get one now, not at the last minute.
- The following must be written clearly on the outside of the envelope:
  - Your name (or names, if a team).
  - The uniqname under which the source code was submitted (this is just you, if you are working alone).
  - The combination of features that you chose (e.g. S1, S2, M1, M2), so that I can easily group the projects together that worked on the same feature without pawing through the documents.
  - Close the envelope and use the clasp to hold it shut, but do not glue it shut.
- The hardcopy deliverables in their properly identified envelope must be delivered to the professor *in person* at times and places to be announced. If you do not deliver these documents in person, I will not be responsible for them.

Important: If you do not follow these rules for submitting your deliverables, I will refuse to accept them. Be sure to find and purchase a 10 X 13 clasp-type envelope ahead of time. Get the specified size; smaller sizes often tear when I add the code hardcopy. See the picture for the type of envelope required; this is cheap and effective; the clasp keeps the contents secure when closed, and it can be repeatedly opened and closed. This might seem mickey-mouse, but it makes a huge difference in handling the projects. So it is a real requirement. If you show up with loose pages of paper, be prepared to run to the bookstore or go dumpster-diving to come up with an envelope.

Project Evaluation

The score for Project 6 will be based only on hand-grading, and unlike previous projects, the bonus award applies to the hand-grading score. A portion of the hand-grade score will consist of manual run-testing of your program as described, but if the program runs basically correctly, almost all of the score will be based on document, design, and code quality.

The autograder is used only as an easy way to send in your soft-copy materials, and because I will be run-testing your code in the autograder environment, the autograder will also do a check compile for you using the makefile you supply. The autograder awards a single point for a successful compile, but that single point is not part of the project grade - it means only "yep, it compiles." The actual project score comes completely from the hand grading.

The score will be based on your specific definition of features, and how well your program meets the design goals, as illustrated by the examples that you implement and as explained by your deliverable documents, and manifested in the structure of the code. To put it positively, a good specification of the features, a good design that is general and extensible, quality code, and clear and informative documents, should result in a high score.

To put it negatively, if your specific features and design are minimal in scope, you will receive a minimal score. An *excellent* way to get a *horrible* score is to write code that will only handle the specific examples that you implemented - you've missed the whole point of the project (see above).

Teams: All members of the team will get the same Project 6 score. The submitted Project 6 code will be evaluated for both Project 5 and Project 6 requirements. Differences in number of features will be adjusted by averaging the scores for each feature together, so the scores for all team sizes will be on the same scale.

Important: Do not implement more than the required number of features. If you do, no "extra credit" will be awarded, and I will base the grading on the *worst* of the features that you supply. The idea is to do a good job on the required number of features that you choose to design and implement, not deliver a hodge-podge of hacked-together code.

Suggestions for getting a good evaluation

- *Write quality code*. This project is to be programmed in Standard C++ and take full and appropriate advantage of the Standard Library, and the usual rules of quality coding apply. Compare the C++ Coding Standards Document against your project code. Since this is the third time the general code quality will be evaluated in this course, any shortcomings in general code quality will heavily penalized. So if you've had general code quality problems in previous projects, you should take extra care with this one. Learning from the previous evaluations is critical.
- *Look for opportunities to apply the concepts, techniques, and design patterns presented in this course*. Do not re-invent the wheel or adopt a clumsy approach when a better one was presented. To put it negatively, if your project looks like you never took this course, or skipped the last half of the lectures, then it will get a poor evaluation. Rather, this is your chance to put these ideas together and apply them to get a great result.
- *Take care with the documents - see the above warning*. I won't bother to puzzle out your code if your documents are not helpful. Thus poor documents will lose credit for both the documents and for most other aspects of the project. In terms of scoring, it will be better to submit buggy code than inadequate documents, so allow time to develop the documents. Drafting the feature description and design documents before you start coding will actually help you work faster and better.