Computing & Software Systems 342: Mathematical Principles of Computing Fall 2011

Basic Course Information

Along with CSS 343, this fast-paced course is intended to bring you up to speed so you can take Junior and Senior level CSS courses. It does this by integrating the fundamental mathematics of computing with detailed instruction in program design. By the end of this quarter, you will be familiar with much of the C++ language and the basics of object-oriented programming. You will understand how to analyze a problem and design a solution. You will know many basic data structures, algorithms, and the tradeoffs among memory, running time, and implementation time associated with them. Topics include: recursion, computational complexity and algorithm analysis, logic, mathematical proofs and induction, lists, stacks, queues, sorting and searching, data abstraction, and object-oriented methods.

Instructor. Michael Stiber stiber@u.washington.edu, room UW1-360D, phone (425) 352-5280, office hours Mondays 10AM–12PM or by appointment.

Course Web. http://courses.washington.edu/css342/stiber/.

Lectures. Tuesdays and Thursdays, 3:30–5:30, room CC1-041.

Textbooks.

- Frank M. Carrano, *Data Abstraction & Problem Solving With C++: Walls & Mirrors*, Fifth Edition, Addison Wesley, 2007, ISBN 0-321-43332-7.
- Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, Seventh Edition, McGraw Hill, 2012, ISBN 0-073-383090.

Suggested Reading.

- Bruce Eckel, *Thinking in C++*, Second Edition, vols. 1 & 2, Prentice Hall, 2000. A good "from scratch" introduction to the language. Available for free in electronic form on-line at `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`.
- Bjarne Stroustrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, 2000. The canonical reference. Make sure you get the third edition.
- Harley Hahn, *Harley Hahn's Student Guide to UNIX*, 2nd edition, McGraw-Hill, 1996. If you're unfamiliar with Unix or Linux and want to learn more, get a book like this.
- Herbert Schildt, *STL Programming from the Ground Up*, Osborne/McGraw-Hill, 1999. A very good introduction to the Standard Template Library, with lots of examples. However, it is not a reference; for example, it doesn't provide complete lists of methods for each class.
- Nicolai M. Josuttis, *The C++ Standard Library*, Addison-Wesley, 1999. Much more of a complete reference than the Schildt book. Includes some examples, but is not intended as a tutorial.

Grading. Your course average is computed as: 25% homework + 10% peer design reviews + 25% midterm + 30% final + 10% online contribution

I don't grade on a curve. I compute everyone's quarter average based on the formula above. I then use my judgment to determine what averages correspond to an 'A', 'B', etc. for the quarter. Some quarters' assignments, etc. turn out harder, and so the averages are lower. Other quarters, averages are higher. I adjust for that at the end. Decimal grades are then computed using the equivalences in the *Time Schedule*, linearly interpolating between letter-grade boundaries. Furthermore, I am well aware of the significance of assigning a grade below 2.0, in terms of impact on your career here at UWB. I can assure you that I examine *in detail* the performance in this course of each student before assigning a grade below 2.0.

What is the difference between this and grading on a curve? With the latter, the goal is to have X% 'A's, Y% 'B's, etc. My way, I would be happy to give out all 'A's (if they were earned). FYI, in a "typical" quarter, below 60% might be a 'D', 60%–75% a 'C', 75%–85% a 'B', and above 85% an 'A'. You may use this as a rough guide; however, if you *really* want to know how you're doing, please see me. *I reserve the right to adjust these scores to reflect the specifics of assignments, test questions, etc. for each quarter.*

Course "Rules"

The following is a brief summary of the most important things you can do to succeed in this class:

- Read the entire syllabus. Every word. Failure to follow instructions (for example, neglecting coding or documentation standards, trying to turn an assignment in the wrong way or after a deadline) *will* adversely impact your grade.
- I will not try force you to use any particular development environment. However, your programs *must* compile with g++ and run on the CSS Linux machines. Please be advised that, if you use any other environment, it is possible that you will spend considerable extra time "porting" your code to the class compiler and computing environment. It is even possible that you will never get your program working. It is a worthwhile investment of your time for you to learn to use Unix and g++. If you use another development environment, then you assume all responsibility for getting your code running on the class computing platform; I will *not* make exceptions and test programs otherwise.
- Similarly, there are certain class standards for documentation, including simple class diagrams. If the tool you use does not produce diagrams that look like the class requirements, please find another tool. Neat hand drawings are perfectly acceptable.
- You are responsible for making back-up copies of your work. Disk crashes, etc. are *not* acceptable reasons for extensions of assignment due dates. Note that your Linux and UW IT home directories are professionally backed up; free services such as Dropbox are also useful in this regard.
- Assignments are due when specified. Barring illness or similar extenuating circumstances, please do not attempt to submit amendments, bug fixes, or forgotten material after the fact.
- While I do not formally require your attendance in class, participation in peer design reviews is a portion of your grade. Additionally, I will assume that you are cognizant of everything that is covered in class, including clarifications of programming assignments, changes in due dates, etc. Material covered in class is fair game for assignments and tests, regardless of its absence from the textbooks.
- I may use email to communicate with you. It is your responsibility to ensure that email to your UW account reaches and is read by you. Note that the UW course listserve will only accept messages from addresses that are on the official list. So, if you forward your UW email, you will still need to send email messages to the listserve from your UW account. According to my experience in previous quarters, your UW email is almost certainly more reliable than any free account (e.g., Hotmail).

Graded Activities

As indicated under "Grading", above, your grade in this class is computed from a weighted average of five different activities during this quarter. You are of course familiar with tests; below I describe the others.

Online Contribution

We will be making heavy use of the course discussion forum to provide a means for out-of-class, in-depth discussion, either of topics I select or of questions/comments by fellow students. I will generate a new thread for each week — some I will kick off with a question or other material to get the conversation started, and some wil merely be available for conversations surrounding the homework, tests, or whatever is going on in class that week. It is your responsibility to make valuable contributions to at least some of the conversations during the quarter (I will place a guide to what makes a contribution valuable elsewhere on the course web site). At the end of the quarter, you will be asked to submit your two best posts as evidence of your contribution; your contribution grade will be computed from those.

As an alternative to a forum post, you may substitute a substantive, non-duplicative entry into the CSS Wiki. Presumably, this entry will cover some aspect of using the CSS computing infrastructure, of workflow hints for inter-operation of your home computing environment with that in the CSS program, etc.

Homeworks

We will have both written exercises and programming assignments. While all programming assignments will have a value of 100 points, the value of written exercises will vary (likely in the range of 15–30 points). Subsequent sections of this syllabus carefully spell out (in detail) both the procedures for program submission and the content of what you should submit. Please *read the syllabus carefully.* If there is anything you don't understand or are not sure about *ask me. I will assume that you have done so, and will mark off if what you submit does not match what is required.*

Programming Assignments

The philosophy behind the programming assignments is to exercise your growing abilities to solve problems using computers. Note that I do *not* say to exercise your programming ability; I assume that, though you may be a beginner, you are basically familiar and comfortable with the process of writing and debugging software. In this class, you will learn about a variety of problem-solving tools: algorithms and data structures (taken together, abstract data types), approaches (for example, recursion), and mathematical techniques to compare and develop new algorithms (algorithm analysis, logic, mathematical induction, etc). The homeworks are explicitly designed to be substantial — they will require you to use what you learn in a systematic manner. As an example, imagine that we have just covered the topic of lists, and in class discussed a list of integers. I would *not* assign a homework that has you implement a list of strings. The point of learning about lists would be to understand them so that you can use them to solve problems (and to know when they are appropriate for use). Therefore, a more suitable assignment would be one in which you are asked to implement a simple text editor, which internally might use a list to hold the lines of text. This would allow you to investigate how a generic abstract data type's capabilities can be related to the specific functionality of a particular program.

Assignments will be due at specific dates and times. I will *not* accept *any* lateness in this class — if your assignment is submitted even a few minutes late, it will *not* be graded, and you will receive a *zero* for that assignment. In fact, we will be using the UW *Collect It* system, which will not accept late submissions. Except for special circumstances, such as medical and other emergencies, *no* exceptions will be made to this policy (this includes crashed/eaten/lost disks — *make frequent backups*). You are more than welcome to submit work before the due date.

We will be dividing assignments into two phases, and you will be responsible for completing each phase by the indicated deadline. Because of the amount of material and the tight timing of the quarter, it is *absolutely essential* that you carefully read and follow the procedures in this syllabus. I will present an initial assignment description during class and will expect you to commence on the assignment immediately. *I make these assignment purposefully vague and incomplete in certain respects.* The two phases are "specification and design" and "implementation"; they are graded separately — the former as part of a *peer design review* and the latter in testing by me.

1. It will be your task during the initial specification and design phase (see "Specification and Design," later) to define the problem to be solved and produce a solution. This will involve identifying where the original assignment description is incomplete, and at the next class meeting, asking questions to clarify matters so you can finalize the specification. Your specification and design will be due *one week after the assignment is handed out.* I will assign you in class into groups for a peer design review, we will discuss the assignment and design issues together as a class, and then you will hand in your designs so you may receive credit. Your name, student number, and email address should be written on your hard copy submissions. Please strive to either write/draw clearly or use a computer and high-quality printer to prepare your documentation; I cannot give you credit for what I cannot read.

   Note that this represents one-half of the time allotted to your programming assignment. Therefore, before handing in your preliminary specification and design, you should ask yourself, "Does this represent 50% of the work I'll do on this program?"

2. At this point, you will have roughly a week to *implement* your design. You will submit your program using the UW Collect It system. The Collect It area will only accept tar or zip files, and the submission size is strictly limited, so please submit *only* source code. I will unpack your submission in a directory dedicated to your assignment, compile all of the `.cpp` files into a single executable, and test it. Testing will be performed using `g++` on one of the 320 lab Linux machines. **It is your responsibility to ensure**:

   - That, when unpacked, all of your files will be present, in a the same directory as the archive (i.e., the archive should *only* contain files, not a subdirectory), and have appropriate capitalization (Unix is case-sensitive; Windows is inconsistent regarding case sensitivity. If I need to manually rename all your files, I will reduce your program grade). I suggest that you test to make sure that your submission workflow is correct by transferring your archive file to one of the CSS Linux machines, unpacking it, checking that all of the files are present in the same directory (and not a subdirectory), and then compiling it and running it.
   - That any long lines in your software are neither truncated nor wrapped.
   - That your program will compile using `g++` on a CSS Linux machine using g++. The course web site and CSS wiki has information about lab 320, Linux, and Unix. `g++` is our course compiler; if you choose to use another compiler, then be forewarned that you have made a decision that may increase the time it takes for you to complete the assignment (as you may need to modify your code to get it working under `g++`). If you make that decision, please be responsible for the outcome — do not expect that I will take into consideration that it compiled and ran under another compiler.
   - That the input your program expects and the output it produces **exactly** matches the specifications in the assignment.
   - That your name and student number are in a comment in each source file.
   - That you follow all other coding conventions outlined below.

I allocate credit based on your coding style, your documentation, and on your program's execution characteristics ("correctness", determined by *black-box* testing). For example, if your program does not compile under `g++`, you will receive *zero* points for correctness. I will run your program against a set of test cases (which I will *not* release ahead of time); partial credit will be awarded if it passes only some of them. I will *not* debug your program or try to figure out why it doesn't work — partial credit only comes from passing test cases. Any other way of assigning partial credit would, in my opinion, be unfair: based more on my debugging ability than the qualities of the program. Because of this, I require you to design your program before you write code, and I strongly urge you to implement your program in stages, so that you always have a partially working version. If you use a development environment other than g++, then I suggest that you periodically move your code to a machine that has g++ to compile and test it (again, I do not endorse the approach of using another compiler, I am merely suggesting prudent practice). Of course, I am more than happy to meet with you about your program before or after the due date.

Program Grading. Generally speaking, adherence to coding standards will be worth 15% and program correctness (in other words, does it work?) 85%. However, depending on the specific nature of each assignment, the exact percentages (and any other aspects' weights) may change. One example of this would be an assignment including a significant written portion.

Special needs

If you believe that you have a disability and would like academic accommodations, please contact Disability Support Services at (425) 352-5307, TDD (425) 352-5303, FAX (425) 352-5455 or at rlundborg@uwb.edu. In most cases, you will need to provide documentation of your disability as part of the review process. More information is available at the DRS web site.

Collaboration

Unless you are specifically involved in a group activity, you are expected to do your work on your own. If you get stuck, you may discuss the problem with other students, provided that you don't copy code from them. Programming assignments *must* be developed and written up independently. You may always discuss any problem with me. You are expected to subscribe to the highest standards of honesty. Failure to do this constitutes plagiarism. Plagiarism includes copying assignments in part or in total, debugging computer programs for others, verbal dissemination of algorithms and results, or using solutions from other students, solution sets, other textbooks, etc. without crediting these sources by name. Plagiarism will not be tolerated in this class, any more than it would be in the "real world". Any student guilty of plagiarism will be subject to disciplinary action. Please believe me, neither you nor I want to go through an academic misconduct hearing.

Class attendance

I strongly encourage you to come to class. You will be held responsible for all material covered in class, regardless of its presence (or lack thereof) in the textbook or web site. Additionally, a portion of your grade depends on participation in in-class activities, such as laboratory exercises.

Problems

If you have problems with anything in the course, please come and see me during office hours, make an appointment to see me at some other time, or send email. I want to make you a success in this course. If you have trouble with the assignments, see me before they are due. If you fall behind, it will be difficult to catch up.

Tentative Course Schedule

Date	Topics	Reading	Assignment
9/29	Welcome; Linux, development tools, & C++; software engineering principles	Carrano, Ch. 1, App. A	Program 1 assigned
10/4	Recursion; Recurrence relations & induction proofs	Carrano, Ch. 2, App. D; Rosen, Ch. 5	Program 1 peer design review
10/6	Recursion as a problem solving technique	Carrano, Ch. 5	Program 2 assigned; Written HW 1 assigned
10/11	Abstraction & OOP in C++	Carrano, Ch. 3	Program 1 due
10/13	Linked list implementation & memory management	Carrano, Ch. 4	Program 2 peer design review; Written HW 1 due
10/18	Stacks: implementation and applications	Carrano, Ch. 6	Program 3 assigned
10/20	Queues: implementation and applications	Carrano, Ch. 7	Program 2 due
10/25	C++ objects, classes, and OO design	Carrano, Ch. 8	Program 3 peer design review
10/27	Algorithm analysis	Carrano, § 9.1; Rosen, Ch. 3	Written HW 2 assigned
11/1	Midterm review		Program 3 due
11/3	**Midterm**		
11/8	Sorting	Carrano, § 9.2	Program 4 assigned; Written HW 2 due
11/10	Sorting, cont'd	Carrano, § 9.2	
11/15	Propositional & predicate logic	Rosen, Ch. 1	Program 4 peer design review; Written HW 3 assigned
11/17	Logic, cont'd	Rosen, Ch. 1	
11/22	Trees	Carrano, Ch. 10	Program 4 due; Program 5 assigned; Written HW 3 due
11/24	**Thanksgiving**		
11/29	Trees, cont'd	Carrano, Ch. 10	Program 5 peer design review
12/1	Mathematical foundations	Rosen, Ch. 2	Written HW 4 assigned
12/6	Math foundations, cont'd	Rosen, Ch. 2	Program 5 due; Written HW 4 due
12/8	Course wrap-up		
12/13	**Final**		

Design and Coding Standards

> "If builders built buildings the way programmers wrote programs, then the first woodpecker to come along would destroy civilization."

> "If cars had followed the same developmental path as computers, a Rolls Royce would cost $100, get a million miles per gallon, and explode once a year, killing everyone inside."

The two quotes above vividly describe the contrast between the typical practice of "programming" and that of other engineering disciplines. What is the difference? Historically, programming was not practiced as an engineering discipline. Practitioners took pride in their ability to hack out solutions. Oftimes, the more elegant solutions were, the harder they were to understand. "If it was hard to write, it should be hard to understand!" was the hacker's motto.

Much has changed in the last couple decades, and one of those changes has been the upgrading of Computer Science education to match that of other engineering subjects. Thus, programming becomes Software Engineering (SE), and Software Engineers spend considerable time and effort on activities other than programming. At first blush, this may seem a waste of time. However, nobody would think that of the time spent by a civil engineer designing a building, or an electrical engineer designing a computer. In those other fields, there is a big distinction made between *design* and *construction*, with the latter often not considered engineering *per se*. The same has been increasingly true of software engineering, with software design being *engineering* and programming becoming *coding*.

The reasons for this are typically couched in terms of dollars, because the largest consumers of software engineers have been corporations, and it is most convenient for them to convert everything into units of money. However, almost all of the arguments made in favor of SE for the corporate environment also apply elsewhere.

The key to to understanding the advantage of the "design first" approach is to consider the entire software life cycle. When one writes code without designing it ahead of time (and therefore without any design documents produced), one is making (at least) *all* of the following assumptions:

1. That only one person (oneself) will ever have to look at the code.

2. That the problem is relatively trivial (that one can keep the entire solution in one's head, down to the smallest detail).

3. That the program will be used once, then thrown away (so one won't have to remember 6 months from now what one did before).

4. That there will only be one user (so no need to refer to design documentation to answer user questions).

If any of these assumptions are violated, then a design is necessary before *any* code is written (except perhaps for some prototyping, though there should still be informal designs done for those):

1. If more than one person needs to write code or work on the design, then a design document is the *only* way to communicate system function. Code is *not* documentation, it is implementation. Code does not indicate the function a program is supposed to perform; it only indicates the function that a program actually does perform. Additionally, code is a set of formal instructions meant for a computer; it is an extremely inefficient way to convey meaning to human beings. Often, it is easier (and perhaps faster) to rewrite code than to understand it undocumented.

2. The solution to any nontrivial problem must be worked out in advance. Systems are often implemented in modules, and interoperation must be assured. One may need to switch one's attention from one part of the system to another, and design documentation is an essential knowledge base for storing what is known about parts one isn't currently working on.

3. Six months or more from now, it can be difficult to remember exactly why everything in the code is there. So, not only is design documentation necessary for communication with others, it is also necessary for communication with future versions of oneself.

4. Ofttimes, users will ask questions about software not answered in whatever user guide is produced. At that point, if design documentation is available, and answer may be easily produced. If the answer so arrived at does not conform to the user's experience with the program, then a bug has been discovered. Therefore, design documentation also helps in the debugging process, allowing one to determine when actual system operation deviates from that which is desired.

> "A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, 'Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world.' The civil engineer interrupted, and said, 'But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth out of chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world.' The computer scientist leaned back in her chair, smiled, and then said confidently, 'Ah, but who do you think created the chaos?' "
>
> — Grady Booch, *Object-Oriented Analysis and Design*

Specification and Design

This involves two parts: determination of the desired system functionality (specification) and the actual design. The former involves major interaction with the end-users; the latter brings to bear CS knowledge (theory, algorithms, practice) and software engineering technique. We go to this trouble for one simple reason: software systems are the most complex objects routinely constructed by people. A thorough, careful design and development process is the only practical way to manage this complexity. As Grady Booch says, "We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems."

Your documentation should be written so that someone else could take your specification and design the program, or your design and understand how your program works (including be able to modify your code).

For this class, you must write a *problem specification* and a *program design.* These will be used in the peer design review in class, and then handed in to receive credit. The process for conducting a peer design review will be covered in class; below I present a description of what your documentation should contain. Note that, as you read the Carrano textbook, this will seem very much like the denigrated *waterfall* software development method. This mostly reflects the fact that I am asking you to design first and check your design in class; it doesn't mean that you won't be iterating back and forth among analysis, design, implementation, and test. In fact, I strongly suggest that you at least take an incremental approach to implementation, so that you always have working code that does something.

The specification makes the problem statement precise and detailed. There should be nothing ambiguous or unknown left after the specification. Your specification should *not* just be a regurgitation of the assignment statement; they are purposely left vague and incomplete so as to require you to go through a process of refining precisely *what* the program should do. Your specification should reflect the results of this phase of your homework, and your grade for the homework will be, in part, determined by the specification. Divide your *specification* into the following sections:

Problem statement In your own words, introduce and describe the problem to be solved. This section should also answer the following questions: What assumptions are possible? Are there special cases? Is there anything unclear in the original problem statement given to you that you clarified with me? Any assumptions that you made yourself?

Input data What is the program's input data? From where will it come? In what format? What data is valid and what data is invalid?

Output data What is the form of the output?

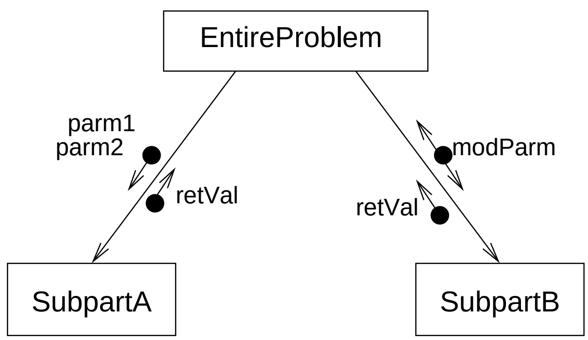
Error handling What error detection and error messages are necessary?

Test Plan Consider the set of possible inputs to your program (defined in the "Input Data" section). Can you break this up into subsets which are similar in some way? For example, if you were writing a tax preparation program, the part that deals with capital gains might treat negative numbers (losses) differently than positive ones (gains). For each of these subsets, choose a small number (perhaps three) of test cases (one good rule of thumb is to use the two *boundary elements* [largest and smallest values] and one *typical value*). For each input, determine what the correct output should be. The resulting table of (input, output) pairs is your *test plan*. Make sure you document the rationale for your test plan; don't just report the test plan by itself. Do *not* just produce a plan that tests erroneous input — your test plan should focus primarily on *testing the correct operation of your program.*

The program *design* document is a complete and unambiguous description of how the program will work. It is language-independent, and includes a description of the overall structure of your code (classes and objects, calling and called functions and their interfaces). This should be the result of your own design process performed *prior to the start of coding*, rather than something done after you've written the program. I expect two main parts to your design: *structure charts* and *simple class diagrams*.

Structure Charts

A structure chart graphically displays the hierarchical algorithmic structure of a system. If we were taking a pure object-oriented approach in this course, then we would dispense with structure charts. In that case, our systems' basic organizing principle wouldn't be a hierarchy of function calls, so structure charts wouldn't be useful. However, in this course, we are merely using classes and objects to encapsulate abstract data types, and so our programs are mainly hierarchical in nature. The figure below presents a simple structure chart.



You can think of a structure chart as a map of your program. It shows how the program is broken into (sub)procedures, which functions call which, and the data they pass back and forth. Each function in your program should be represented in the chart by a rectangle with the function name inside (if the function is a method of some class, then use the C++ notation `className::functionName`). Lines connect calling procedures to called ones, with arrows indicating the direction of the call. If a procedure is recursive (calls itself), there is no need to indicate this in the structure chart. These lines are annotated to indicate parameters passed and values returned as follows:

"Pure" parameters These are either passed by value or as `const` references; they are not modified by the called procedure. Indicate these by a dot with an arrow pointing to the called procedure. Label them with the parameter name(s) — *use the names the calling procedure uses.*

Return values Indicate these by dots with arrows pointing back to the calling procedure, labeled by the name the calling procedure uses (if the result is stored in a variable) or (if the value is used only ephemerally, as the return value of `pow()` in `a = b + pow(c,d)`) not labeled at all. If the return value is not used by the calling procedure, then omit it.

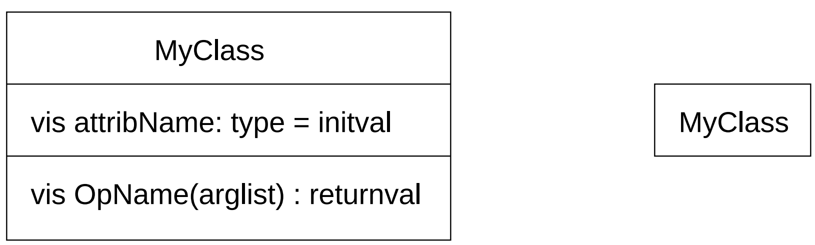
Modified parameters These are parameters passed by reference or to which pointers are passed, and which are possibly modified by the called procedure. Indicate these with labeled dots that have arrows pointing to *both* the called and calling procedures.

You use structure charts in your design by starting with the entire problem to solve — this is the top box of your structure chart. Break this problem into major subproblems, and write down how you could solve the entire problem if you could solve the subproblems. This gives you enough information to add the subproblems to the chart as procedures, with properly annotated calls. Repeat this process for each subproblem until you've reached problems that you can solve directly. The resulting structure chart and notes is your structured design. Your design documentation should include *both* structure charts and the pseudocode you wrote during the above decomposition process.

Simple Class Diagrams

Class diagrams define the static structure of an object-oriented system (systems with a number of interacting objects will also have a dynamic, run-time structure, but we will leave this to CSS343 or later). The figure below shows two versions of a simplified UML (Unified Modeling Language) icon for a class: one with detailed information (left) and one without (right). *You should only draw one of these in any one diagram; for this course, that will always be the one on the left.* A class icon is a rectangle with the class name inside (an icon for an object has the name underlined, so *don't* underline the class name). The detailed view is broken into three parts:

1. The class name (top).
2. The attributes (middle). Each attribute has a name, an optional type, and an optional initial value. Each attribute is prefixed by its visibility: public (+), protected (#), or private (-).
3. The operations (bottom). Each operation includes the complete signature of the C++ method and its return value, if it is a function. As with attributes, operations can be prefixed by visibility indicators.



As you might guess, UML supports more complex specifications for individual classes, as well as ways of specifying relationships among classes. We will incorporate these into our documentation as the class progresses (so, you have something to look forward to).

Design Standards

You are expected to adhere to certain basic principles of good design:

Variables Each variable (whether it is a primitive type, a composite type [such as an array], or an object) has an associated *scope* and *storage class*. A variable's scope may be local to some small block of code (e.g., a loop), local to some function, "global" within some file (using the `static` keyword), or truly global throughout your program (possibly limited to some `namespace`). A variable with local scope may be allocated at block entry and deallocated at exit (storage class `auto`) or be allocated at the start of program execution and exist across block executions (storage class `static`). You need to decide the scope and storage class of each variable in your design. *You should use non-local and static variables only if they are truly needed, and global variables only if absolutely necessary.* You need to justify your design decisions for all non-local and static variables.

Functions A function (this includes methods) should perform a *single, simply describable operation.* If you find that a "function" you are considering really does two things, then it is probably better to make it two functions.

Parameters and return values One reason for the above definition of functions is that their interfaces are kept small — they have fewer parameters and return values. Monitor your functions' interface complexity. If you are passing/returning many items, this may be a sign that this is not a function.

Methods One of the major topics of this course is that of abstract data types (ADTs) and their implementation using classes. Remember that the functions you define for your ADTs will correspond *exactly* to the methods in your code. For each method (actually each ADT member, be it an operation or a data item), you will need to decide whether it should be publicly accessible or not. *You should make a method public only if that is truly necessary, based on the definition of the ADT in question.* Your design should make the distinction between public and non-public methods very clear, with the public nature of methods self-evident from the definition of the ADT.

ADTs and the implementation "wall" ADTs consist of an externally-visible interface and a supposedly hidden implementation. However, a clever programmer can circumvent the wall around implementation by returning internal, implementation-dependent information about the ADT. You *must not* do this — it goes against the purpose of object-oriented design.

ADTs and UI/IO ADTs implement internal data types which are independent of the exact nature of any particular program. *ADTs should not include any user interface operations.* As a simple example, imagine you are designing a vector ADT, and that you included operations tied to a graphical user interface. This would mean that your ADT would be *unusable* in a non-GUI environment, such as a computer controlling a car's engine. You should detach issues of user interface and I/O from ADT design — they are *separate* parts of the design. Note that it *is* acceptable to include generic stream I/O operations in your ADT design (implemented in C++ with overloaded `friend operator<<` and `friend operator>>`), as these are fundamental and machine-independent.

Coding

Coding standards means writing code that is easily understood and including comments that clearly document its function. Code clarity is aided by consistent and useful indentation, identifiers with descriptive names and naming conventions, and the use of special language constructs, such as `const` and `typedef`, which you will learn this quarter. More precisely, our course coding standards are:

Indentation Blocks of code should be indented three spaces. This includes the bodies of functions.

Variables Variables should be given descriptive names, unless they are very clearly just loop counters or the like. There should be comments associated with each variable declaration explaining how the variable fits into the algorithm, and including invariant information such as its legal range of values.

File comments Each file should begin with a comment containing the file name, author name, date, and a description of the purpose of the code it contains. The file that contains `main()` should also include documentation for the overall program: a description of the program's input and output, how to use the program, assumptions such as the type of data expected, exceptions, and a brief description of the major algorithms and key variables. This is the information you generated in your design, *before* you started coding. It is expected that you will merely copy the appropriate sections of the your design document into comments for each file and function (see function comments, below).

Class files Separate `.cpp` and `.h` files should be used for each class. The file names should match the class names *including capitalization*.

Library includes C++ STL include files should be included like `#include <vector>`, rather than `#include <vector.h>`. Similarly, C includes should be as `#include <cmath>`, *not* `#include <math.h>`. It is acceptable to use the directive `using namespace std;`.

Classes Do not return references or pointers to internal class/object structures. Classes *must not* expose *any* of their internal implementations.

Functions Functions should be used for appropriate modules, with *reference arguments* used only when necessary. The *type* of each function must be declared (use `void` when necessary).

Function comments Each function should be preceded by a comment with a short description of the function's purpose and precise *preconditions*, *postconditions*, *return value*, and *functions called.* For methods, this comment should appear in *both* the `.cpp` and `.h` files.

Loop invariants Each loop should be commented with *invariant* information.

Assertions *Assertions* should be inserted into the code where useful to explain important features or subtle logic. You may use comments or the `assert()` feature for these.

You are also expected to avoid global variables (and you are required to justify their use if you do need to use them) and will *not* be permitted to use `gotos` in this class.