CMPT 275-d1 (Semester 02-2) **ASSIGNMENT #4**

DUE: At the START of class, Friday, July 19, 2002.

In this assignment, you are to do the low level design of, and write all the modul/class implementation code. You must be able to compile every module, but it doesn't have to link. Second, you must run a single unit test case of your main module using stubs, and also run a single unit test case on one of your more complex bottom layer procedures using a driver (both these WILL require linking in either stubs or a driver with the code to be tested. Note that Java uses dynamic linking that is done automatically).

Don't forget to do all your coding according to the team's coding standard developed in Assignment #3 and recorded in a large comment in your main module. Also, I am always shocked when students write a pile of code which, come Assignment #5, doesn't generate outputs exactly as described in the user manual. I can't figure out why students would not write code according to user specifications in the first place, rather than coding 'creatively' and then having to change either the manual or the code so they agree. Therefore, all coders should have a copy of both the user manual and the coding standard in hand while coding!

Note 1: Getting a program to print directly to the printer is not easy in a sophisticated operating system like WinNT. Therefore you may want to write your flight report to file, and in the user manual tell the user what file to manually print out.

I want you to implement the persistent storage needed using fixed-length record binary files. These are used in almost all databases, so as to be able to do fast random access (i.e. to get the 100th record instantaneously, just advance the file a distance equal to 99 times the length of a record, then read a block of bytes into a record/struct/class instance variable. Binary records are to be used because there is no use converting integers, et cetera, to characters when storing to file, then having to change them back to 2's complement when reading from file. Be careful not to use class instances if they have or inherit virtual functions because there is a hidden pointer attribute in each such instance that you do not want to store. Java always has this). You will likely want to read the sections of R. Tront's Cmpt 212 lecture notes on binary, random access file I/O that are in the course notes section of the 275 web pages (advanced.io.pdf), or the file on Binary IO for Java (BIOJava.rev.11.pdf).

To save you effort, I will allow you to do only very crude file organization for the data files. Instead of B+ trees, or even binary search of sorted records, you may use unsorted records. Retrievals can use a linear search (which is fairly fast if there are only a few records in each file, as in your prototype tests). To add a record, append it to the end of the file. To delete a record, find it, overwrite it with the contents of the last record in the file, then truncate the file so it is one record shorter. DO NOT TRY A FANCIER FILE ACCESS METHOD! If you want to later change one of your larger files to do a fancier implementation in Assignment #5, TAKE MY ADVICE and get the unsorted implementation working first. Then, if your performance is poor, and IF YOU HAVE TIME, and if your design is 'well hidden', you can simply change the one module which adds, changes, deletes, and retrieves records from that file! This is exactly why we do design info hiding: so maintenance (i.e. enhancement and performance upgrading) is easier. Besides, recall that inserts and deletes are VERY SLOW in sorted record files, even though access is fast. By the way, you are NOT to pre-read entire parts of files into RAM in order to work on them there. You must make the assumption that users will have later entered so many items that there would not be enough room in RAM. Second, you are to open and close the files only ONCE during the execution of the program. No professional programmer opens and closes a file for each read, or even for each submenu. You are NOT to try to avoid implementing a shutdown scenario by opening and closing a file for every access, or upon each entry/exit of a sub-menu. And, at the end of every scenario, you must make sure the disk files are up-to-date such that if the power fails at that point, no data will be lost!

I will hand out some info on truncating/shortenning a file and seeking to file positions in C++. We will look for some way to truncate a file in Java. If this is not possible, Java teams don't have to bother to shorten the file, however you then have to somehow have a way of knowing where the last valid record is.

The module interface design was completed in Assignment #3. The detailed design documentation for Assignment #4 will be composed of two parts which augment that done in Assignment #3.

The first part will be a Detailed Design Document which contains a written discussion of any detailed design issues affecting/shared **by more than one module**. If you have some structure, say, whose inner details are shared by several modules, you could comment/document this structure in one of the modules and in comments of the other modules point the programmer to the one module it is commented in. Or, if you do not want to document this in one particular module, you can put it in the Detailed Design Document. Even if a design concept concerns only one module, you may need diagrams to indicate how it works, and diagrams are hard to draw as internal comments. So again, this kind of information could be put in a Detailed Design Document.

When discussing the detailed design (either in the Detailed Design Document, or in source code implementation comments, GIVE HINTS as to where weaknesses are, where detailed implementation (not architectural) speed or memory inefficiencies

are, and where things are that might need to change if the system is ported to a different computer or operating system, etc. During Assignment #4 you should not discuss any high level architectural trade-offs, just ones relevant to coders/maintainers of individual implementation modules, or of sets of modules that share a design secret.

The detailed design documentation should discuss the design options/trade-offs that you had to make with regard to issues like algorithms, hidden data representations (types and structures), and file organization. Generally, in the low level design document itself, you only need to discuss those issues which are relevant to more than one module (modules on rare occasions do share a data structure or other design secret), issues which are of an overview nature (e.g. "all persistent object storage in this design ..."), or for diagrammatic reasons are impossible to put in the source code as comments. Other comments/ discussion related to only one module should go as comments in the source code itself, and need not show up in the design document. I anticipate the Cmpt 275 Detailed Design documents can be very short as much of the detailed design information can be put in the various modules! Note that design decisions which have already been described in the high level design need not be repeated here. But if some of your decisions require that you revise your Architectural or Interface designs, you must revise and re-submit your Architectural Design Document.

You must also include why you chose a particular implementation detail VERSUS some alternative. Even if the instructor mandates a particular choice (e.g. unordered fixed-length records), see if you can describe it's advantages. You will loose marks if you do not mention at least one ALTERNATE choice for each major low level design decision, the alternative's advantages and disadvantages, and WHY you chose each design trade-off you did. This goes for both the decisions described in the design document, and for those mentioned in the source code comments! Feel free to praise the good aspects of your low level design (algorithms, structures, representations, storage or performance, etc.).

Most documentation of the design of individual implementation modules/classes and function algorithms can be put as internal comments in the implementation source code files themselves, rather than in the document Each implementation module will need a revision history at the top and then an introductory comment which augments the corresponding header file's overall comment. There is NO NEED to re-describe in the implementation module what kind of functions the module offers, as any programmer will have already read the header files. Instead, the implementation module's introductory comment should describe any *overall* design issues internal to this module. Comments about how individual procedures are implemented should go after each procedure signature, but comments about design issues which most of the procedures share (e.g. pop() and push() share the fact the stack is implemented as a linked list) should go at the top of the module's implementation or class. Also, comment each non-exported module or static class variable declared in the module. e.g. "pointer to first element in stack list". By the way, don't use the term 'global' variable when you mean it is a module variable. The more comments that you can put in your implementation module source files (even crude diagrams), the less that need be written externally.

Each C++ function implementation in the .cpp file should begin with a dividing line comment, the procedure signature, and then an implementation comment describing how it is implemented. Note the interface comment for an public/global C++ function is in the header file; do not repeat it in the .cpp file. Note that the header file comment might say a function finds and returns an object, whereas the implementation module comment might say it uses binary search algorithm. And AddReservation() might be commented to say it adds the record at the *end* of the file. Also, each function should have comments in its executable statements. At minimum, each loop must have a comment describing the ***goal*** of the loop. The step size of the loop should be mentioned (e.g. searching forward by bit/byte/word/record/level/node?). Conditionals in IF statements and loop ends should be understandable either by well named variables in clear conditional expressions, or with a comment. Compound conditionals always need a comment. Don't forget to comment any local variable declarations.

Java will be a bit different because the interface comments for a function and the implementation comments cannot be put in separate files. I would suggest putting interface comments (those useful to a calling programmer) after a dividing line and before the function signature using javadoc style. And put implementation comments after the signature. e.g.

```
/**----------------------------------------------------------
  *function reserve:
  *{interface comments here for calling programmer}
  */
public static void reserve( int id){
// {implementation comments here for programmer who might have to edit this function body}
{
   reserveForID( id);
}
```

You needn't document the procedure parameters in a .cpp file as to direction and units as that is already done in the header file. On the other hand, if you start your implementation module by editing an copy of the definition module, I wouldn't bother removing the in/out and units info. But you should remove the header file revision history and general module/class comment, because the implementation modules have their own revision history, and their comments describe HOW, not what the module and functions do.

Also, remember your audience. You are going to write code, internal comments, and external documentation, then are leaving for England on a long posting. Your best friend, who was not involved in the analysis or design or coding, is going to have to test, debug, and enhance your code. Assume (s)he has read the previous documents, but needs to understand and modify the code. How would you comment this for your best friend?

In either the detailed design document or in the code comments, you may also want to mention reasons why certain representations were chosen for numerical data. For instance, if the range of data is greater than 64k, you should choose 'long' rather than 'int', because if the software is ever ported to a 16 bit OS, most PC compilers use 16 bits of storage for simple ints. You also may want to discuss a data structure (e.g tree vs. list) and related algorithm (e.g. O(log N) vs. O(N/2) search performance) and why you chose it relative to the anticipated relative frequency of operations (e.g. random and sequential access vs. insert/delete) or the size of a file (e.g. unordered lists are ok if files are very small). You may end up stating that the implementation choice for this first prototype is inefficient, and make recommendations for later improvements. Occasionally, some groups do weird projects in 275 and if they use any unusual structures (e.g. linked lists, trees, etc.), they must supply data structure diagrams ("a picture is worth a 1000 words"). If your project this semester doesn't have any unusual structures, then of course you won't need such diagrams.

All your code must compile, though you needn't and probably shouldn't link it. The code you write is NOT part of the detailed design document, but each code file is a separate Computer Software Configuration Item (CSCI) that is mentioned in the release table. Put the source listings after the Detailed Design Document. Each module should be separated by a labelled-tab divider, though you needn't put dividers and tabs between a C++ header file and its immediately following implementation file. The modules should be in the order main module first, then the rest in alphabetical order. (They should be in alphabetical order in the release table too, except main at the top!). With many compilers, you can ask that a line numbered listing of the code be produced with error messages and statistics (like number of warnings and total number of statements). This could be handed in to prove your file compiles without errors. Unfortunately, there doesn't seem to be any way to get Borland C++ to generate such a listing, so you will have to hand in a floppy and the TA will randomly choose a few teams to test compile. Hand in any Borland or Visual C++ project files that might help him, or Jbuilder or Forte for Java project files.

UNIT TESTS

In addition on this assignment, I want you to write and execute one unit test case that requires stubs and one that requires a driver. These tests are simply to allow you to get first hand experience with the concept of unit testing, with what stuff is needed in a stub to get it to barely link and work, and what is needed in a driver. It will also give you an insight into incremental integration. The unit tests can be either black or glass box, as you now know the implementation details.

You must write and run one unit test case for the main module using as many supporting stubs as necessary. Do you remember the 4 parts of a test case? In the test case, point out to the reader via the overall OCD which modules will, as a result of communicating with the main, have to be stubbed. This test need not test the whole main, but it should test something you are unsure of. Because it may not need to make all calls that the main could make, the stub modules need ***not*** have fully functioning bodies for all functions which they export. Some functions can have totally empty bodies if the main will not call them during this test case. Remember, stub modules are needed for compiling and linking, and stub procedure functionality is only needed to run those portions of the main that will actually be tested. Generally, you will use the same header file for the stub as for its full implementation. The stub implementation module should have a similar file name, but obviously a lighter implementation. In some languages, when an interface/.h file in turn includes other interface files, a tree of imports can happen, thus requiring even more stub implementation modules. This likely will not happen in C++, but if you are using another language, you might want to comment out unneeded imports in a stub's interface file.

And you must, via a properly written test case, unit test one procedure in one low level module using a driver. The tests need NOT be extensive. Nor must you thoroughly (100% branch) test even a single procedure in a bottom level module (though you should state the % branch coverage that your unit tests cover). Just do a single test case run on one procedure. Remember, a good test has a high likelihood of finding an error! A SUCCESSFUL test FINDS errors. Don't spent time looking for them in places where they are unlikely to be. Test a low level procedure that you are not sure will work. I will be slightly disappointed if you DON'T find and document an error, as you will not have used your test time productively toward getting a bug-free project. You will not loose marks if you find or don't find an error; the point is do a sensible test which probes for an error(s). One of the categories of tests you may want to try is whether a low level procedure, which in turn calls the operating system (especially binary record files), work as you initially expect. Design your driver to stimulate your low level procedure to make calls to the system supplied routines which you are MOST unsure of. Do NOT fix any bugs you find in your application's code, but point them out in the test result. Only your driver and stubs must work well, so as to be able to actually run the unit tests in order to report if a couple of portions of the application code work. (Optional: you may optionally fix bugs found during unit testing and hand in revised source code files. But the test report must specify that the failure was on a specific previous revision number of a problematic module; and mention that the problem cannot be seen in the revised source code that you are handing in. I.e. the unit tests were run and in some cases failed on Release 4, and you are handing in Release 4A. In that case, you would need two, not one, new columns in your release table for Assignment #4. Also, your test case results should specify the revision of modules the test was run on.).

You should hand in copies of the stubs and drivers, probably as a unit test appendix of the detailed design document (rather than separately with the rest of the listings). I will not require you to mention the stubs and drivers on the release page either. Also note that some students find that during unit test creation, they realize that to test one procedure they have to assume another is working. If you AddRec to a file, the only way to check it got there is to call GetRec (and hope it too works), or to dump the file with a utility. I will (somewhat unsafely) allow you to assume the GetRec procedure works OK as it is unlikely both procedures would be in error.

Do NOT try to get your system as a whole running and start playing with it and debugging it. This is not required for Assignment #4. Spend any spare time you have getting Assignment #4 complete (e.g. help or proof-read other team member's work). I ask this because the first time you link the entire system, I want you to immediately run the 3 black box test cases you wrote in Assignment #3. I don't want you to do ANY whole system debugging prior to running the black box test cases, as I want there to be a high likelihood they will fail (you will not lose marks if they fail). On the other hand, you may want to have one person doing special advanced unit tests that you don't even bother handing in for Assignment #4; perhaps binary file I/O or cargo loading algorithm. Only if you absolutely want to start using the whole system for special testing should you link and run, but if you do, immediately record the results of the black box test system test that you authored in Assignment #3.
Note that in Java, there is no such thing as static linking. Java does some checking at compile time that C++ does during static linking. And Java leaves the actual linking to be done 'dynamically' during execution of a program. Generally, do NOT bother to run the program.

BUILD PLAN
In previous semesters, I always asked students to provide a build plan. You are not going to be doing a build, but I wanted the plan in place. This is easy on Borland C++, as you can ask the IDE to create a makefile for you. You could add this to your Release table as part of the release. If not using Borland C++, I had asked students to draw a build plan in a kind of DFD format. However, since I won't be asking the Java programmers to do this (the java compiler when running on each module,does some make analysis and building but never for the whole program), I guess I should not ask this of the C++ students either. If anyone can find a way to get JBuilder or Forte for Java to output something like a make file or make plan, I would love to hear about it.

MARKING GUIDE:
You will be marked on the sensibility of your low-level design (you may rave about your modules or function if you want), and properly justified algorithmic and data representation design trade-offs (e.g. simplicity, speed, memory usage), on programming style, consistent naming and indenting conventions (see the third assignment), module and procedure implementation comments, and comments within the algorithmic code itself.

5 marks - Rel. Title page, Rel. history and table, consistent revisions in other documents and code. If the Architectural design has changed since Assignment #3, state why in the revision history of the Architectural Design Document.
5 marks - Discussion - any broad project-wide implementation issues and trade-offs, data and file structure diagrams if they are complicated, and discussion (if not in module or procedure code comments).
8 marks - Sensibility and elegance of design
5 marks - Executable statement coding coding style and quality.
4 marks - Implementation Module introductory comments + revision history, memory/speed/complexity trade-offs, representation choices and alternatives, and if necessary in-source-code crude diagrams. Also, comments on private/static module constants, types, and variables.
4 marks - Function implementation introductory comments, and comments on local constants, local types and local variables. And comments on loops and compound conditionals.
10 marks - Test cases - descriptive overview of stubbing with respect to the OCD, 4 parts of cases plus statement of branch coverage, documentation of exact results, and pass/fail decision. Similar for driver.
5 marks - Spelling and Grammar
4 marks - Organization, neatness, compliance with coding standard, dividers between alphabetically-ordered revisioned items.
50 Marks - Total

Appendix on File Truncation

File truncation is usually not easy because each operating system does it different, and therefore it is hard for any programming language library to provide a standard way of doing it. Some operating systems do not provide this feature at all.

In Java 1.2 and later, the way to do file truncation is use the RandomAccessFile class function called:

```
setLength(long size)
```

where size is the number of remaining bytes that you want to have in the truncated file.
In Java 1.4, there is another way for files that are not of type RandomAccessFile. However, you should probably not use it because as of 2002/07/05 I am not sure we have version 1.4 in the lab and thus you will thus not be able to demo your program properly to the TA at the end of the semester. Nonetheless, I provide it for completeness of this discussion. In the package java.nio, try FileChannel.truncate(long size).

If you are using C++, there are other ways to do this depending on which I/O library you are using.
If you are using #include <fstream.h>, also include <io.h> and try:

```
int handle = fio.rdbuf( ) -> fd( );
chsize( handle, filelength(handle) - sizeof Reservation);
```

where fio is the openned file instance. For a file that stores fixed length reservations, this shorten the file by one Reservation.

The above technique does not work for the newer #include <fstream> (note the missing .h in the angle brackets in contrast to the one above). In that case, you need to:

- close the file remembering where the end of the last record is.
- open it using the C function called fopen( ) from <io.h> to get a handle.
- use chsize( ) in any way that you want.
- close the file using fclose( )
- reopen it using your <fstream> calls.

Yes, I know this is aweful.

Finally, if you are ONLY ever going to us a 32 bit windows OS or NT/2000/XP command prompt (not DOS and not unix), apparently (I have not tried this) you can include <winbase.h> and use SetEndOfFile( ).

For any of these functions, you probably shouldn't use them unless you look up the proper documentation on them. EVERY function in these support libraries has been well documented by the authors. Do not assume that all this information is in you 101 book, or in any tutorials I give you. As a programmer, you have to be able to look up the official documentation on the functions in these libraries. See the Sun Java web site for details of every class and function in Java. See the Microsoft Visual Studio or Borland C++ IDE help and reference material for information windows functions, and see the unix 'man' pages for info on using these functions.