CS 202 EFFIECIENCY AND GDB WRITE-UP FOR PROGRAM 2

BY TEJAS MENON; ASSIGNMENT BY PROFESSOR KARLA FANT

This program was in particular a very interesting one, as we as students had the opportunity to test the maintenance, reusability, readability and effectiveness of our code between two sessions of very similar programming assignments. We created an underlying planet class that randomly assigned itself reasonable values in a placeholder DLL data structure to test its implementation and in program 2, we took this implementation (changing it to a CLL) and essentially used it to produce a spatial rendering of each mass body. My realization and assumption for this program is that the more maintainable and expandable our code was in program 1, the easier it became to modify in program 2. For example, having a simple yet effective design layout for classes that should have been 'containing' versus 'is a' versus 'using' brought about a natural style of implementation- each class derived from what should have been it's parent- without incentives for non-object oriented design such as multiple inheritance. Minimizing the amount of functions that required specific derived objects in place of more common ones definitely brought about code that was very reusable between classes- my declare_life(), update_pos() and find_force functions definitely has more scope for usage in terms of the possible areas of expansion for this program. The misc (miscellaneous) class as well is one that can be a superclass to every type of mass and still see some relevancy in its functionality. In this case, it provided operations supported for both spaceship and planet- two very distinct masses in terms of functionality. My goal for this class was to support two way operations- it shouldn't matter if I'm calling the find_force() function from planet or spaceship, both should find the force exerted on its specific body using variances in the argument list (makes sense since both objects are masses, and every mass certainly exerts a gravitational pull on another). Using inheritance relationships additionally brought to the plate ease in implementing mathematical concepts within my classes- certain operations can only be performed on certain classes, but there are common operations that can be performed on all classes. Having said thay however, one aspect of the program that felt redundant/less necessary was compulsory dynamic binding, since only a few of my classes required accessing functions from the derived class while having base class pointers. This I could say was due to the simplicity of our current program- there is only one instance where a class has two derivatives and also my derived classes (rocky and gas) didn't require any of its variables to be modified, and so not having extensive dynamic binding/virtual functions was tenable.

The data structure chosen for the program was also appropriately sufficient for performing a variety of possible tasks with a limited size galaxy and solar system. For example, an array of solar systems allowed for an easily implementable body of code with traversal techniques that were equally easy and a CLL of planets for each solar system allowed for easy resizing at runtime. However, such a simplistic solution for more complex operations, such as in a case where each solar system would require reordering/positioning/shifting, would have been difficult and inefficient to perform on an array. Additionally, as in a real world application, if the galaxy sizes chosen had to be

several magnitudes larger, it would have been difficult to store such a large array contiguously in memory. An alternative would have been to store solar systems in a BST/balanced tree, arranged by their position in space or likelihood of life in a meaningful way. This would have made the process of accessing particular solar systems easier upon expansion of the program.

I did not need much use of any of the debuggers this program as the data structure assigned was relatively simple, with the pathways of execution being very limited- most of the functions were called at construction and the scope of errors were limited to easily rectifiable syntax errors. I did however use Valgrind to correct a few minor character array deallocation errors, which I could pinpoint due to its detailed line number provisions. On one occasion too, I resorted to using gdb to run through specific functions for incremental implementation, and put together working modules of the code very incrementally. The get_type() function was also initially a root cause of several errors right across the program, which I discovered later was due to un-freed memory and incorrectly allocated char pointers. Valgrind is I tool I found especially useful off late with large scale data structures as I could test several implementations of a particular structure and determine the most reliable form of implementation. Also a tactic I found that was time-saving was to note all the different line numbers pertaining to different errors –memory or otherwise- and then make changes on the ones appearing too often before figuring out the rest. This allowed me to prioritize my attention depending on the ‘fatality’ of the error and therefore make continual progress.