Cody Adams

CSC375-01 prog02

Winter 2009

Program 2 Analysis Report

After completing my version of prog02, I reviewed the program requirements once more before turning it in. Although that was not the first time I saw the bit forbidding STL classes, it was the first time I questioned my original (false) interpretation. After obtaining clarification, I tried to use a provided linked list class but ran into problems templating the code to work with my existing code. To avoid any further lateness penalty, I am submitting my only working version, which fulfills all other requirements of the assignment.

My implementation of prog02 has three source files: prog02.cpp, change.h, and change.cpp. prog02.cpp contains the interface and references to change.h and change.cpp. change.h declares the *change* data structure class to store the change information generated by the makechange function. There are also two output functions, one designed for individual elements from a STL list container of *change*s, to be called inside of an iterative loop. The other can iterate any float list passed to it, creating an index of sorts, expecting a finite number of values, but able to be easily adapted for other uses. A previous implementation used an overloaded << operator, which was ultimately commented out, but can still be seen in code.

The data retrieval process is seemingly hidden under several nested loops and conditionals, which maintain precision and help conserve memory usage. Typical file error handling can also be found here.

while ( ! infile.eof() ) // read values

{

…

float adjustedamount = floor( amount \* 100 );

makechange(adjustedamount, result, summary);

…

}

Repeating until EOF, this code takes the data from the file and transforms it into an integer for precision. The *makechange* function is passed the individual value and the two lists mentioned above. The lists are passed by reference so they may be populated. The greedy algorithm used in *makechange* attacks the data in a brute force method, starting with the largest possible currency and taking as many of that denomination as possible, iterating through each possible currency value until the value is zero. The code used to achieve this goal executed in less than a second per item in all of my tests, time to process an entire file was directly proportional to the number of items within the file. The complexity of the change required for each individual item had the greatest effect on runtime. Because my deepest loop nesting is two loops (one nested inside of another) and change is calculated on-the-fly line by line as it is read in by the program, the runtime for this program is very short. Processing a file with 18,000 entries of 9999 requiring 1,926,000 total pieces of change took just over 22 seconds. Processing a file with 18,000 entries of 89.94 requiring 288,000 total pieces of change took just over 41 seconds. . Processing a file with 18,000 entries of 1 requiring 18,000 total pieces of change took just under 6 seconds. As an entry requires additional different types of change, the time required to calculate the change for that item increases. This can be seen where almost 2 million pieces of change took about half as long as less than 300,00 pieces. I believe this is because each while loop within *makechange* is in ϴ(n) [with each iteration perform within being ϴ(1)]. As additional loop is executed, the runtime is increased by another factor of n. To this effect, 20 items requiring 20 total pieces of change takes 10 ms, where 20 items requiring 60 pieces of changes takes 20ms. For the same number of items, the time required varies as a factor of the quantity of change returned. In the worst case, $89.94, 16 loops will be execute, yielding a runtime of 16n.