Jonathon Harris
CS202
Karla Fant
10/16/20

Efficiency Write Up – Project 2 – Account Manager

For this program I had a linear linked list of users. Each user contained an account head pointer that can branch off with a linear linked list of Account pointers, which was my abstract base class. Discord, email and slack were all derived from account and they all had a message node pointer to a linear linked list of messages. When I read the instructions, I interpreted the send message requirement as sending it between users, so because of that my program is rather large. Every account function that was implemented must first find the user in our list of users, then we go through the list of accounts associated with that user and use run time type identification to figure out which account function to call.

The way my program is setup first the client must add a user, then they add accounts to that particular user, if they just try to use the account functions without adding the accounts to the user, there will be an error. The account functions I chose were add message, send message, remove message, remove all messages and read all messages. As far as the relationships, user node was derived from user. Inside user node we had a pointer to the next user node, and an account node head pointer. Account node had an account abstract base class pointer in it as well as a pointer to the next account. I found this structure made sense, if I would change one thing it would be that user node contained a user inside of being derived from it since it had very little interaction. Each class had a clear job to do though, I had a user list class that was in charge of managing my user nodes, the user nodes were responsible for managing the accounts associated with that node. The account node was just a container and had very little jobs to do. As far as the efficiency of the code, it was a lot of replicating functions, for every display, remove, send, and add message function I had to write three of them. In the future it may help to write templates so the function data types can be swapped out.

I have the tendency to do all the implementation of functions first then test everything later. So the first time I used gdb and valgrind after I was finished it was alerting me of a multitude of issues. Because I had three linear linked lists branching off of each other, memory leaks were a huge problem. Valgrind told me in which functions these memory leaks were happening and also alerted me that "a conditional jump or move depends on uninitialized values." So I set breakpoints at the troubled function and stepped through it with gdb. I found that my destructors for my user node and account node were not implemented correctly. So I had to go back and fix them. I still wasn't able to fix the conditional jump problem and that seems to be an error I get in most of my projects. Another issue I had was adding an account to a user, but once I stepped through the code I found out that I had returned null before entering in the add function so that It ended prematurely. In conclusion this program taught me the powerful capabilities of RTTI with downcasting its very significant that you can create a data structure filled with abstract base class pointers that can point to anything in our inherited chain.