Lab Assignment 8

Linked Lists and the gdb Debugger

Zhenming Yang, Connor McEleney

[yang.zhenm@northeastern.edu](mailto:yang.zhenm@northeastern.edu), [mceleney.c@northeastern.edu](mailto:mceleney.c@northeastern.edu)

Instructor: Prof. Zagieboylo

[d.zagieboylo@northeastern.edu](mailto:d.zagieboylo@northeastern.edu)

Due Date: November 9th, 2023

Submit Date: November 9th, 2023

**Abstract**

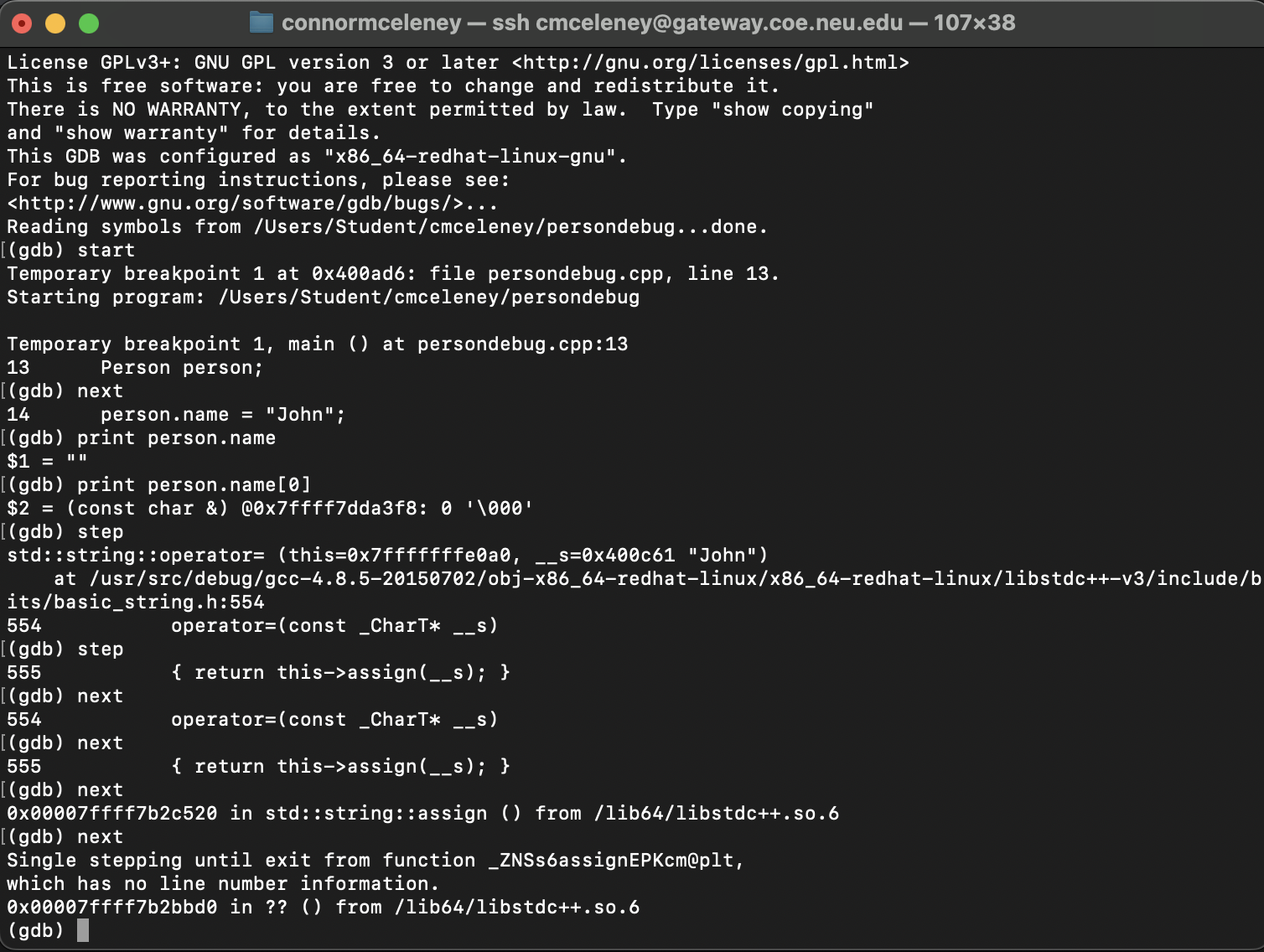
In this lab, we created a linked list object in c++. The program allows user to add, insert, delete, and print the list with person objects in it. We also used the gdb debugger to find the problems and created breakpoints to do code analysis.

# Introduction

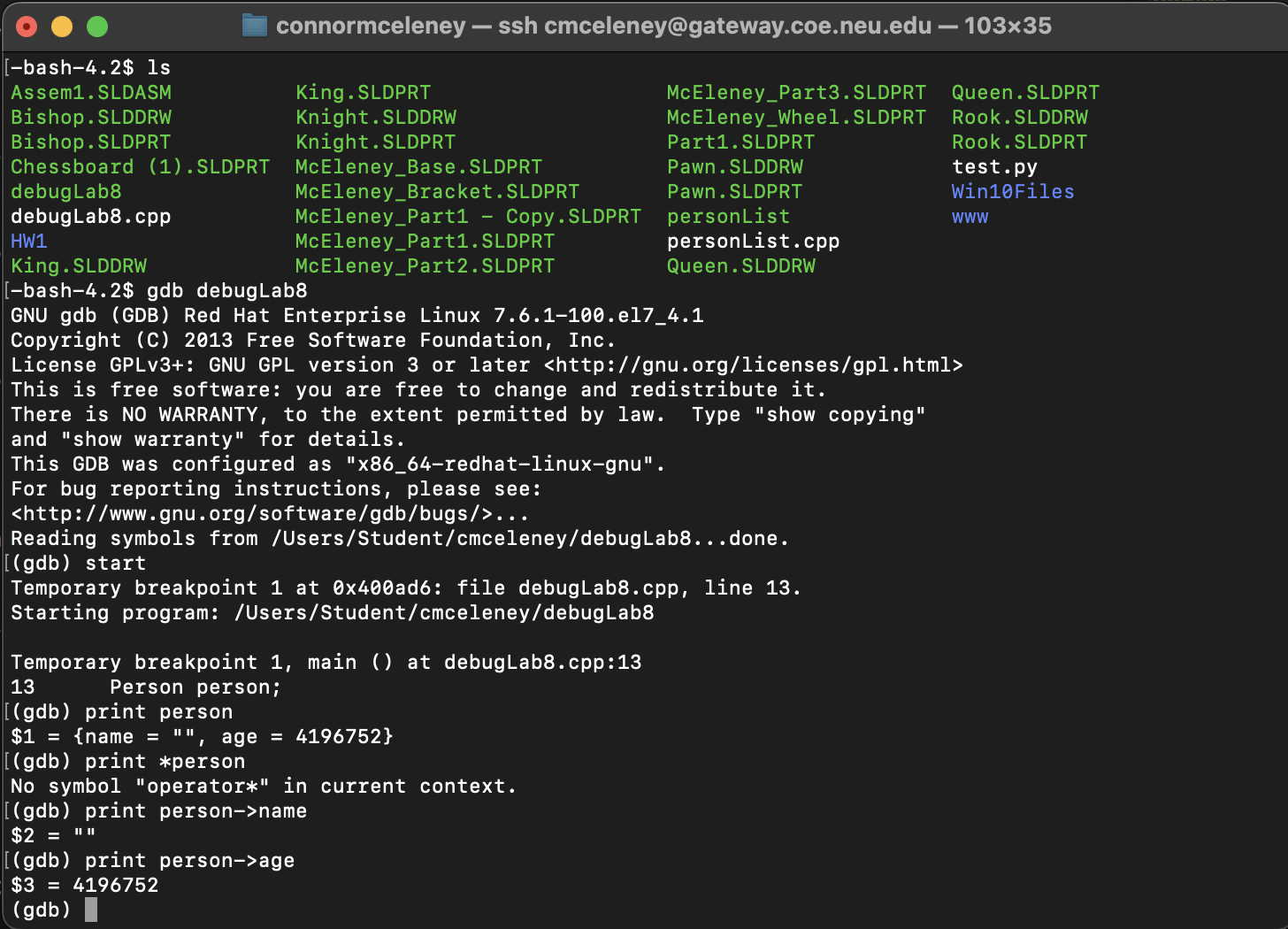
In this lab, we'll engage in hands-on exercises utilizing gdb as a tool for debugging programs. The focus will be on employing step-by-step execution and inspecting memory. Additionally, we'll persist in our exploration of linked lists as an alternative data structure for storing sequences of elements. This structure ensures that insertions and deletions come with a constant cost. Towards the end, we'll utilize gdb to delve into the execution of a main program that employs linked lists.

# Lab Setup

## Pre-Lab



^ Run/next/step ^



^ Breakpoint ^

## Equipment

DE1-SoC:

* The DE1-SoC is a hardware design platform built around the Altera System-on-Chip (SoC) FPGA. The DE1-SoC is designed for experiments on computer organization and embedded systems. It includes embedded processors, memory, audio and video devices, and some simple I/O peripherals.

# Results and Analysis

**Results**

## Part 1: Linked list management

In order to implement the linked list object for storing our person objects, we need to understand what a linked list is. A linked list is a sequential arrangement of data elements, and their order is not determined by their physical locations in memory. Rather than relying on the memory layout, each element in the list indicates the location of the next one. This data structure comprises nodes that collectively form a sequence.

We are given the skeleton code personList.cpp to work with. The given code already provided the required methods to initialize and free the memories of linked list object.

Building upon the code provided, we first created a user interface for the user to input their desired options and the program will interpret it and run the specified functions (Figure 1).

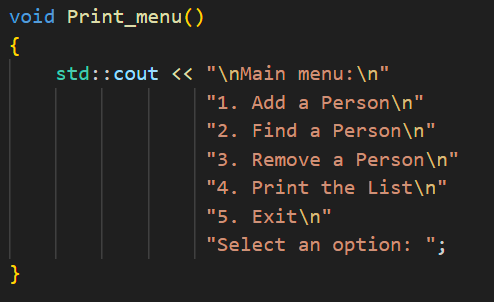


Figure 1: The UI for the user to choose which action they want the program to execute.

The option 1 first called the createdPerson function which prompts the user for inputs to put into the person object (Figure 2). And the function returns the person objects created on the heap and it is then passed it in to the listInsert function that addes the person to the end of the linked list (Figure 3).

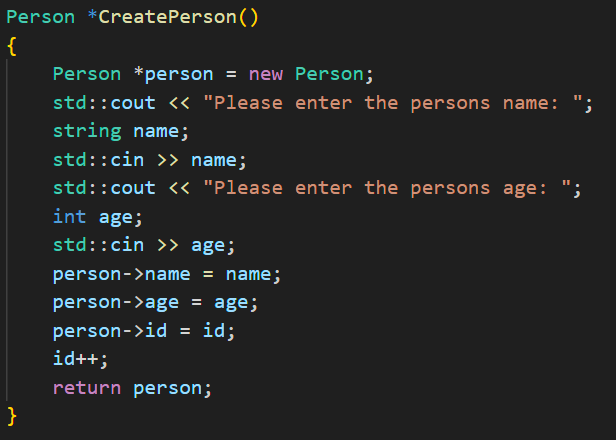


Figure 2: CreatePerson function that takes user input and assigned the inputted value to the person object. Noted the person object is created on the heap and passed as a pointer to the main function.

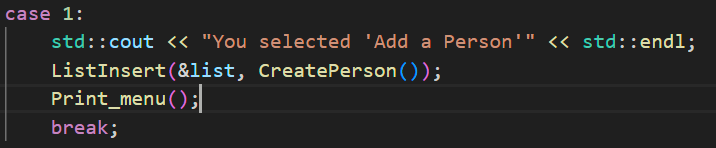


Figure 3: The newly created person pointer is passed into the list insert.

For option 2, we created the FindPerson function to first prompt the user for the id of the person and then loop through the linked list by going through each connections using the give ListFind function and get the value of the person object using the ListGet function. The result is then passed into the PrintPerson function to output the result to the user (Figure 4).

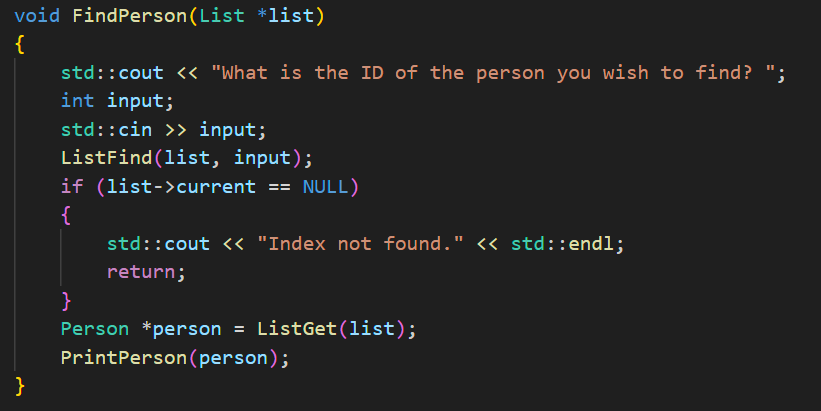


Figure 4: The FindPerson function print the user specified ID person on the terminal.

Option 3 removes the person object in the linked list with user specified ID. We created a RemovePerson function that prompted the user for the ID and then use Listfind to get the object. Then the object is removed form the list using ListRemove (Figure 5).

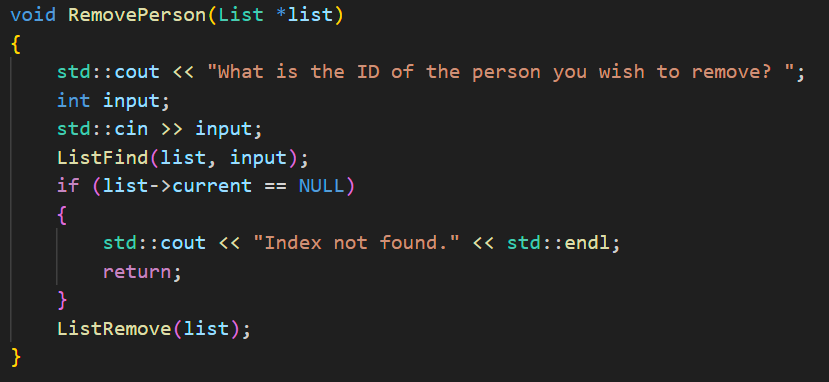


Figure 5: RemovePerson remove the user specified person from the linked list.

Finally, to print the list, we used a while loop to check if the current pointer is at the end of the list. In the while loop, the function first gets the current pointer and passed into the PrintPerson function to print out the first-person object in the list. Then we used the ListNext function to move the current pointer to the next index (Figure 6).

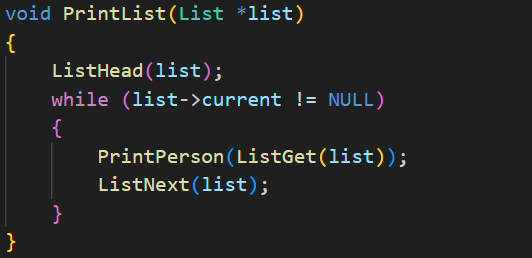


Figure 6: The PrintList function iterate through the list using ListGet and ListNext.

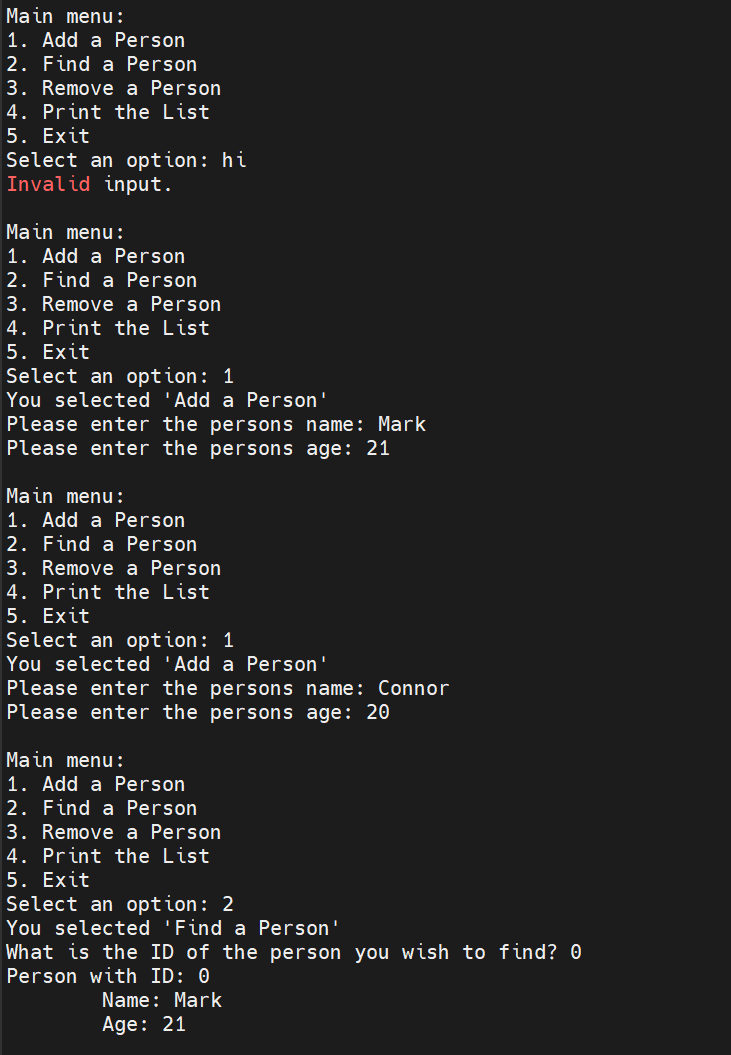
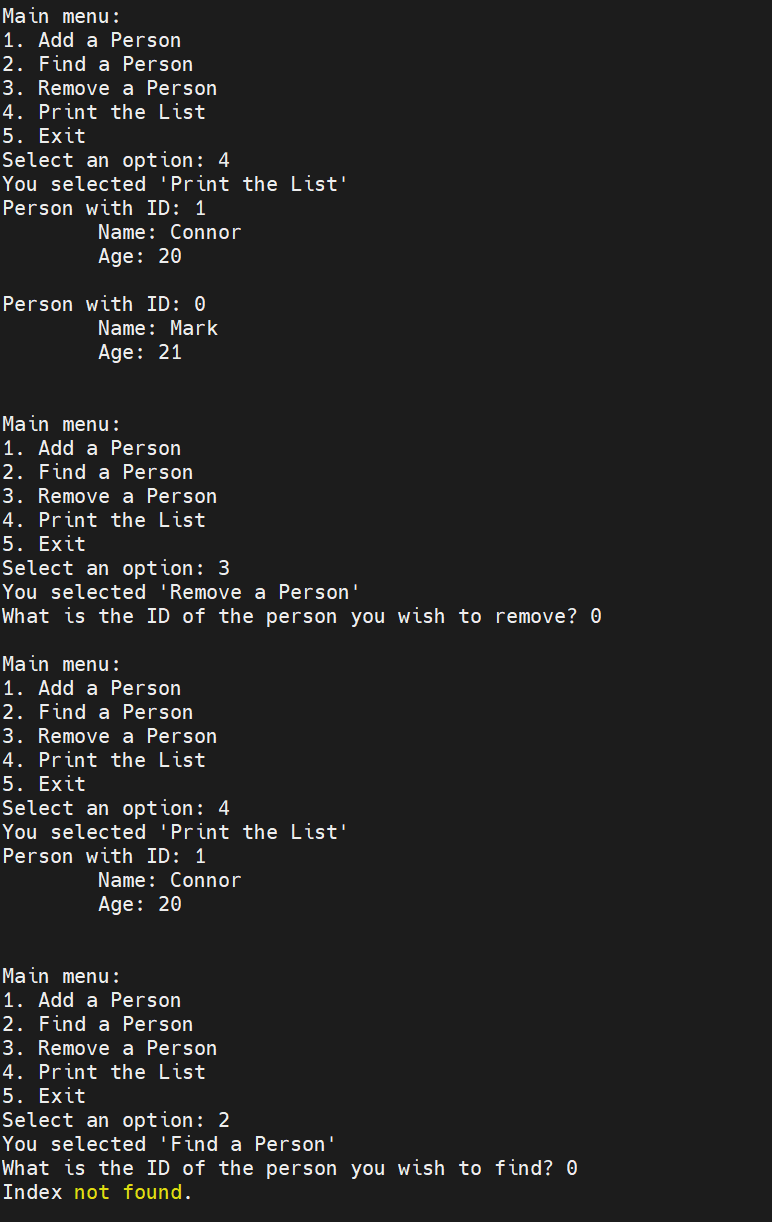
We tested our program on the COE Linux gateway, and it is giving us the correct responds (Figure 7).

Figure 7: We first tried to input string into the option selector, and it gives us an error as expected. Then we added two person objects into the list and we can find the newly added person with id 0. We then printed the entire List out and removed Mark from the list and print the list out again. The output are correct and when we want to find Mark in the list, the program output “Index not found.”

We then run the gdb debugger to see if the link list is implemented as expected. We first created a breakpoint after we add our first person. And then we print out the linked list, the current pointer, and the next pointer of the current pointer (Figure 8). The output is given as expected.

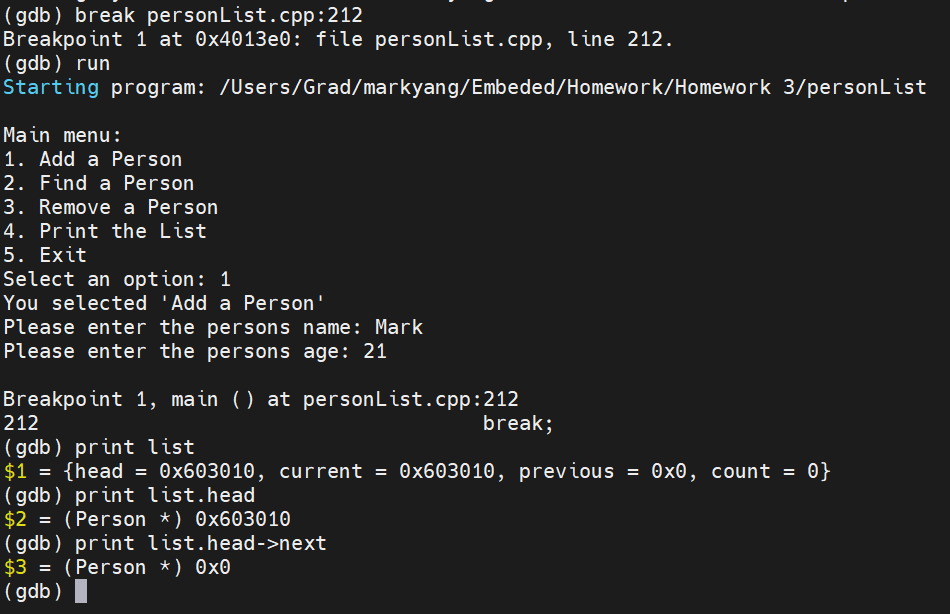


Figure 8: The debug breaks after we added our first person. The head is the same as the current because the head is the only object in the list. The previous is empty and so is the next pointer for the head object.

We then tested the memory operation error we can get from the unavailable memories. We modified the code to when the ListGet function gets something it returns the previous instead of the current. When we print the list with only one person in it. The previous is empty and it will give us this error (Figure 9)

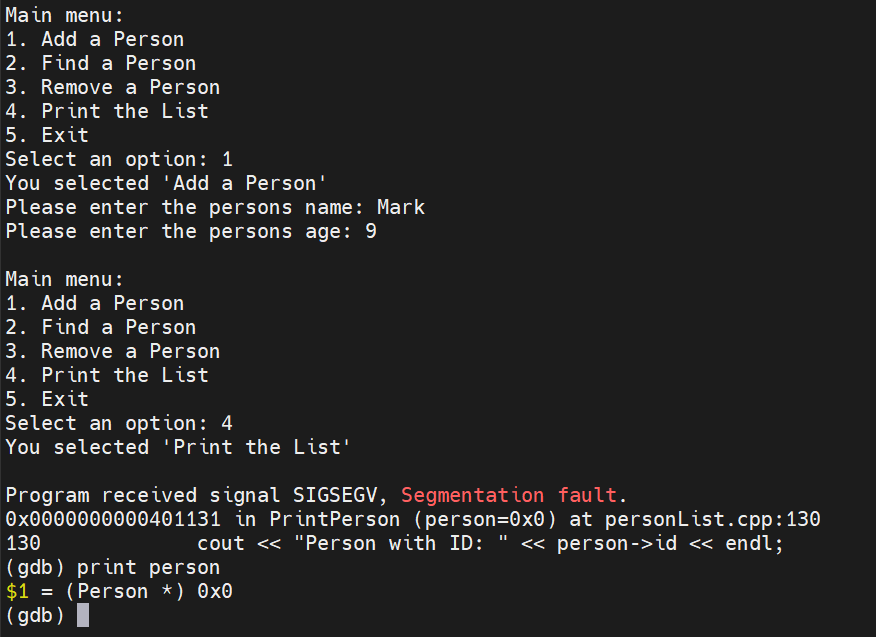


Figure 9: The person pointer is points to an empty memory that can't be read and print it to the terminal.

**Analysis**

This code used a switch statement skeleton to allow the user to easily choose a specific function. The data of the person(s) added was stored into a linked list, a structure that contains the object and a pointer to the next member of the list. The person data structure was referenced with the creation of every new person object, insuring the memory allocation in the heap and its implementation to the linked list. If an index was invalid, such as trying to access a non-existent index at the end, appropriate error messages are printed. The new/delete commands were utilized as better and safer means of memory allocation/deallocation than malloc/free. When exit is selected the program will terminate as expected with exit code 0.

# Conclusion

This lab introduced us to the concept of linked lists and object-oriented programming in C. The means of sifting through and appending to a linked list were explored. We learned the advantages and disadvantages of linked lists, mainly how in linked lists appending an element to the list is very computationally cheap compared to increasing the memory size in an array like we did in the previous lab, however this makes searching through the list because for a non-doubly-linked list you need to start at the first person and follow the pointer trail to the person you want to find since in memory they are non-contiguous and not randomly assessable like arrays. This comparison builds off the last lab in which we constructed such arrays. This lab also served as good practice with pointers (dereferencing/passing out of functions) and structs (how the linked list was constructed)