Lab Assignment 2

Jack Fenton
fenton.j@husky.neu.edu
Michael Delaney
delaney.m@husky.neu.edu

Submit date: 1/30/2020
Due Date: 1/30/2020

Abstract

This laboratory experiment serves as an introduction to formal debugging. This is done using the program gdb, which takes over the process of running the file. In addition, structs are introduced, the first step towards OOP, which are used in a linked list. To demonstrate all of these concepts, a program was written that used a linked list to store data about different people.

Introduction

The purpose of this lab was to introduce the students to gdb, a debugging software. gdb is used to run programs in multiple languages and allows users to run a program line by line, print variable values at any point, and skip forward to predeclared points in the code. gdb is an older program, which is no longer supported on some systems, so it must be run on a computer that has all the required permissions. gdb is also an independent program with its own syntax, so it is helpful to read up on this syntax before attempting to use it, though keeping the documentation open can be helpful. The program itself introduced the composite data type `struct`. A struct is similar to a class, as they are both data structures that contain other data, both data and variables. `struct` was used in this lab for the purpose of linked lists.

Lab Discussion

The prelab assignment consisted of two simple assignments. The first was used to introduce the usage of gdb, named `person`, and the second program set up the menu options for the lab assignment, named `personList`. The results of the first program are shown below in figure 1. The menu options are shown below in figure 2.

```
Temporary breakpoint 1, main () at person.cpp:20
20              Person person;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x86_64 libgcc-4.8.5-39.el7.x86_6
4 libstdc++-4.8.5-39.el7.x86_64
(gdb) next
22              person.name = "John";
(gdb)
23              person.age = 10;
(gdb)
25              PrintPerson(&person);
(gdb) print person.name
$1 = "John"
(gdb) print person.age
$2 = 10
(gdb) step
PrintPerson (person=0x7fffffffe320) at person.cpp:13
13              cout << person->name << " is " << person->age << " years old \n";
(gdb) next
John is 10 years old
14      }
(gdb)
main () at person.cpp:26
26              return 0;
(gdb)
27      }
(gdb)
0x00007ffff7210505 in __libc_start_main () from /lib64/libc.so.6
```

Figure 1: Screenshot of person.cpp run in gdb

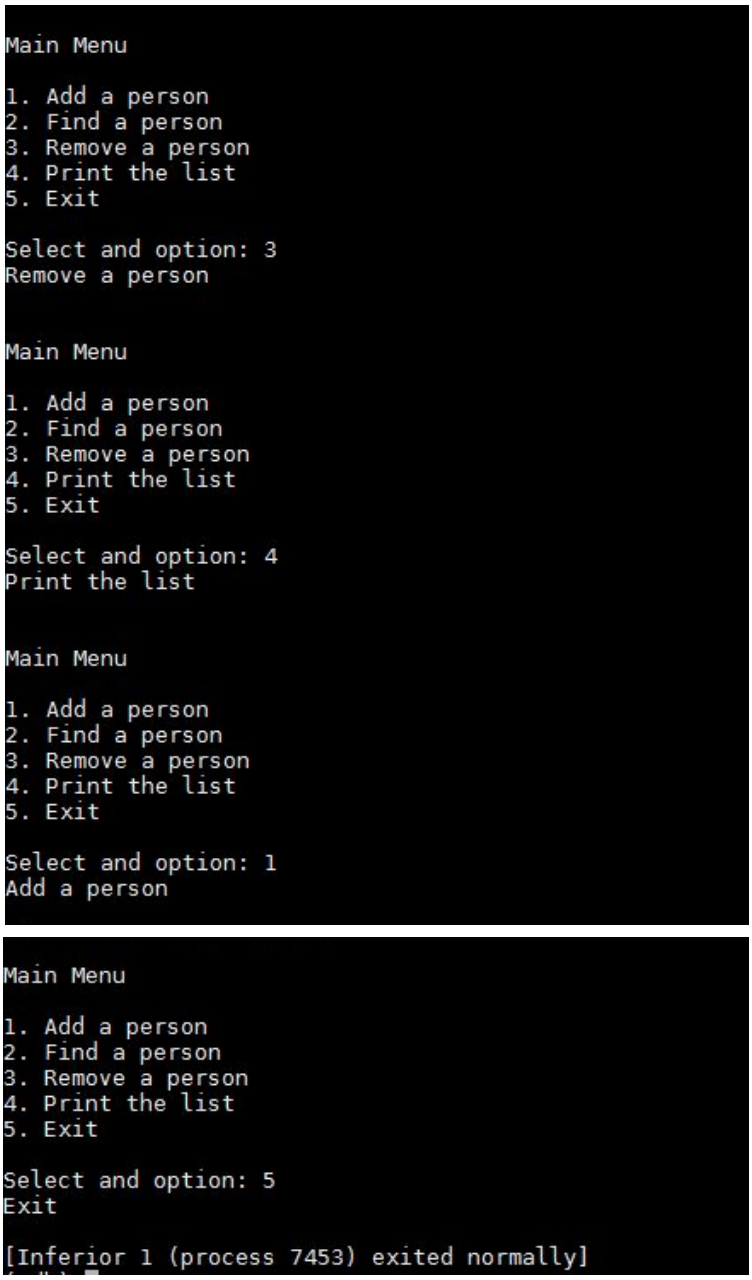

Figure 2: Screenshot of menu options in personList.cpp

After a user selects any one of the options in the list, the selection made is printed to the screen, and the user is free to make any other selection. None of these selections, with the exception of `Exit`, contain any functional code. When the option `Exit` is selected, the program terminates.

During the lab, a COE computer running MobaXterm, and a USB flash drive were used to create compile and transfer the program. Once in the system, the program was created using `vim` and compiled as an executable `.out` file. Using the MobaXTerm software, SFTP was used to transfer the C++ programs on to a USB flash drive that was plugged into the COE computer.

The difficulties in this lab came largely from the usage of gdb. The Darwin Kernel has determined that because gdb requires high level control over the OS, that it has too much potential for malware. Because of this, Mac computers can no longer run gdb scripts without user written keychains overwriting this, even though it is downloaded. This meant that gdb could only be tested on COE machines. This, in addition to slower development using `vim`, led to issues in completing the debugging portion of the lab.

Results and Analysis

Lab 2.4 Breakpoints

Breakpoints were implemented in this lab when running the debugger `gdb`. The purpose of these breakpoints was just as their name suggests: break the code at a certain point. Stopping the code from running at a particular point can be useful in debugging the code because one can then maneuver through the stack at that point. One can also use a breakpoint to stop on a function so they can run through it step by step in order to understand what is happening.

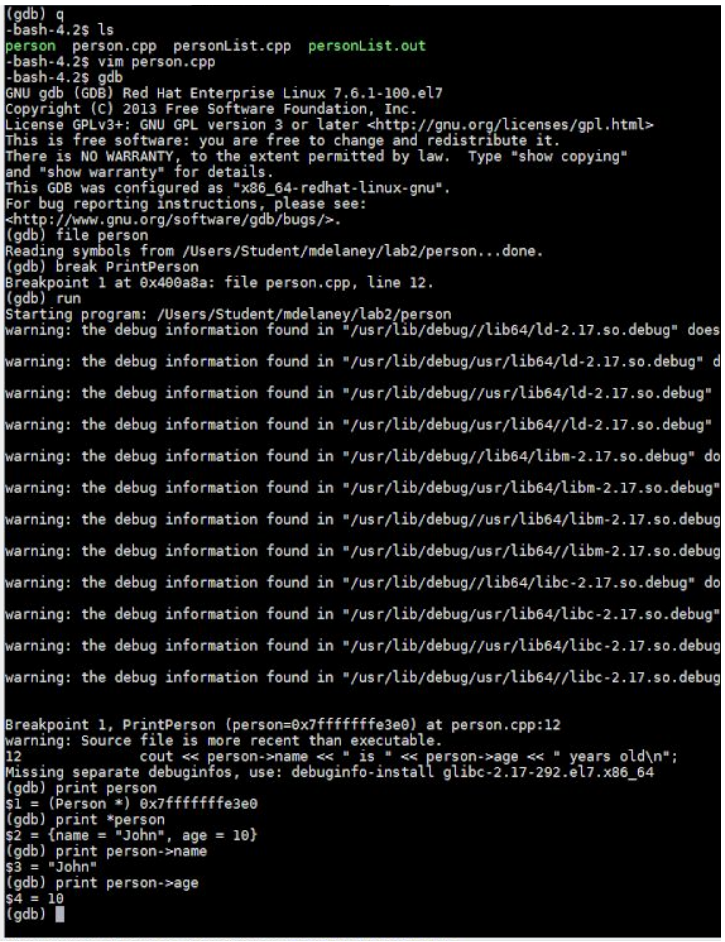

Figure 3: Screenshot of person.cpp run using gdb

The `print person` command above prints the information stored for the variable `person` at that point in the code. Because `person` is a pointer, an address is returned. The `print *person` then returns the value stored at that address, which is the name and the age of the person. `print person->name` prints the specific information stored in the structure under the variable `name`, which is just the name of the person. Similarly, `print person->age` prints just the age stored under the variable `person`.

Lab 2.5 The Stack Trace

The command `backtrace` allows one to navigate the sequence of function calls with the main function. Additionally, one can navigate the stack with this command and see the values stored by local variables at a given point in the program. This can be very useful in understanding why a program may not be working as intended because the precise location of the issue can be pinpointed using this technique. The `up` and `down` commands are used to move up and down the stack of memory and the current progress of the program.

Lab 2.6 Linked List Management

The full code for the file `personList.cpp` can be found in the Appendix. Below is the full terminal output from an example run through the program

```
Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 1
Add a person

Person's Name: Wyatt
Person's Age: 19


Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 1
Add a person

Person's Name: Regan
Person's Age: 34


Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 1
Add a person

Person's Name: Isaac
Person's Age: 51
```

```
Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 1
Add a person

Person's Name: Jim
Person's Age: 26


Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 4
Print the list

id: 1 Name: Wyatt  Age: 19
id: 2 Name: Regan  Age: 34
id: 3 Name: Isaac  Age: 51
id: 4 Name: Jim  Age: 26

Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 2
Find a person

id for information: 2
Person with ID: 2
          Name: Regan
           Age: 34
```

```
Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 3
Remove a person

id to remove: 3

Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 4
Print the list

id: 1 Name: Wyatt  Age: 19
id: 2 Name: Regan  Age: 34
id: 4 Name: Jim  Age: 26

Main Menu

1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit

Select and option: 5
Exit
```

Figure 4,5,6 (left to right): Screenshots from running personList.cpp

As the directions instructed, the execution of the program brings up a menu with 5 options: add a person, find a person, remove a person, print the list, and exit. Options 2-3 will not be able to run without first adding someone to the list, so option 1 is selected first. This calls the `AddPerson()` function.

```
void AddPerson(List *list)
{
   Person *newGuy;
   newGuy= new Person;

   newGuy->id=list->count+1;
   newGuy->name=inString("Person's Name: ");
   newGuy->age=inInt("Person's Age: ");
   cout << endl;

   while (list->current!=NULL) ListNext(list);
   ListInsert(list, newGuy);
   list->count++;
}
```

Each person's data is stored in an instance of `Person`, so dynamic memory is allocated for the new person. Then the person's three pieces of information are set to them: `id` based on the people already in the list, `name` and `age` come from user inputs. The linked list is then traversed until the program has reached the end, where what had been the last element is made to point to the address of the new person.

`FindPerson` is run when option 2 is selected

```
void FindPerson(List *list)
{
   int findID;
   do
   {
   findID=inInt("id for information: ");
   if (findID<=list->count) break;
   else cout << "This is not a valid id \n";
   } while(true);

   ListFind(list, findID);
   PrintPerson(list->current);
}
```

The user is continually asked to input an id for one of the people in the list until the id they input is valid. Once a valid input has been given, the `ListFind` function is used to set the current position to the one that information should be retrieved for. The `PrintPerson` function then outputs the information corresponding to that id.

`RemovePerson` is run when option 3 is selected

```
void RemovePerson(List *list)
{
   int findID;
   do
   {
   findID=inInt("id to remove: ");
   if (findID<=list->count) break;
   else cout << "This is not a valid id \n";
   } while(true);

   ListFind(list, findID);
   ListRemove(list);
}
```

`RemovePerson` has the same structure as `FindPerson`, however instead of printing the person's information they remove them from the list. This is done using the `ListRemove` function, which takes the `next` from the element about to be removed, and assigns it to the one that precedes it, effectively removing it from the linked list chain. Then the memory allocated to that person's information is freed.

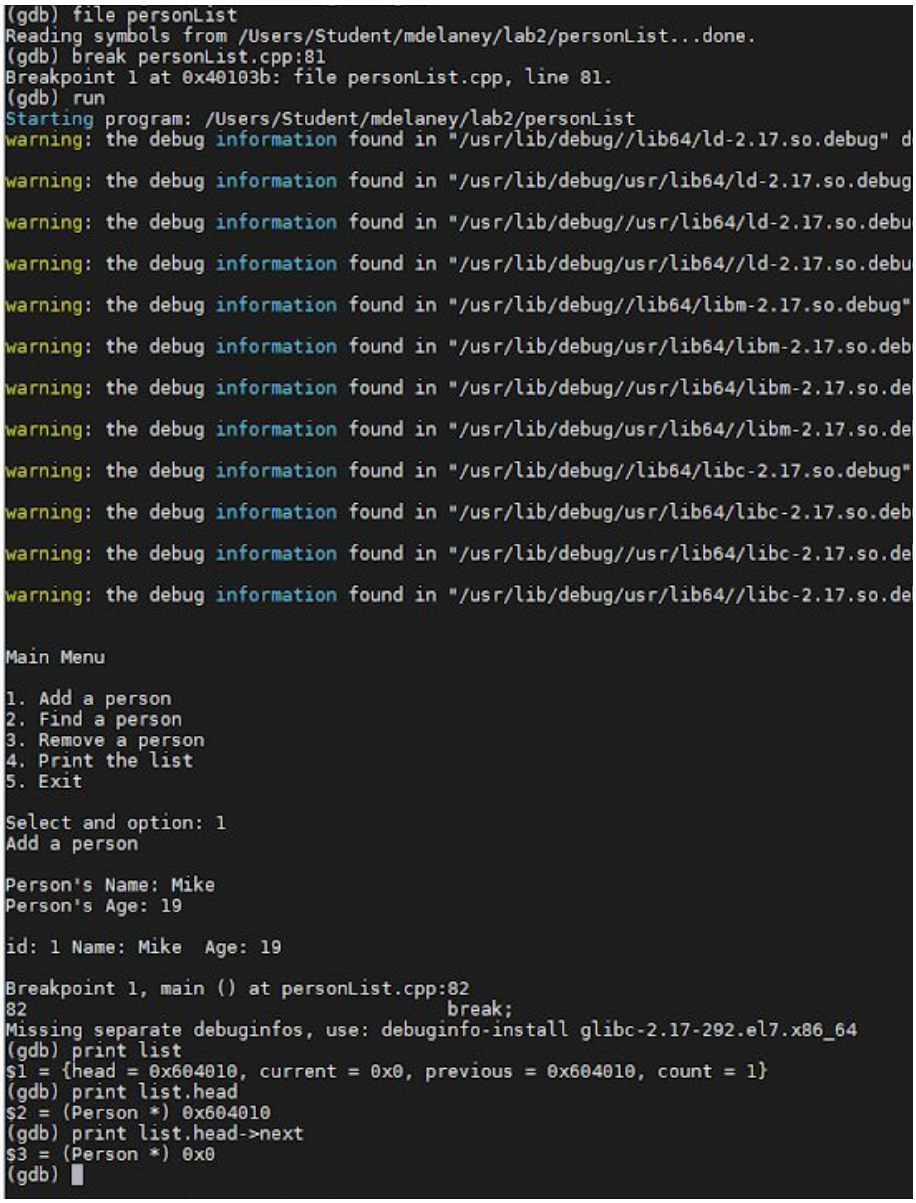

Figure 7: Screenshot of printing using gdb for personList.cpp

The output of the `gdb` reveals the contents of list. This includes the `head` pointer, the `current` pointer, a pointer to the previous value, and the integer `count`. Printing the head returns the `head` of the linked list. Printing the value of `list.head->next` returns the `NULL` pointer. This is the correct value because there is not a second element in the list yet, and this is how linked list should end.

Lab 2.7 Debugging Program Crashes

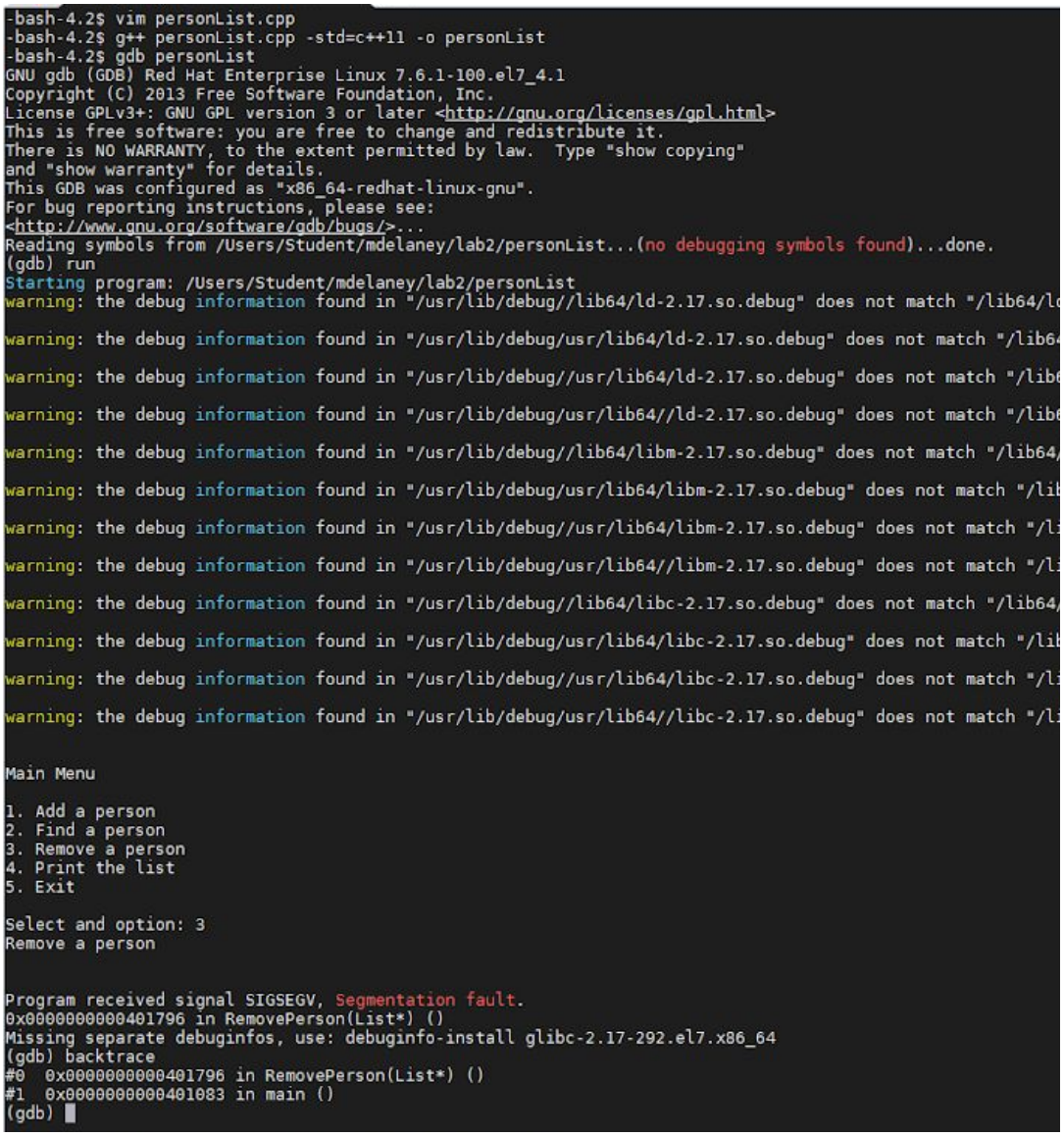

Figure 8: Screenshot of gdb output for invalid memory operation

Running the program through `gdb` returns a segmentation fault. After this, the `backtrace` command is used to look at where the problem occurs. Because no specific variable is named as the cause of this issue, it is known that the arguments from the function `RemovePerson()` are causing the fault (which is accurate).

As the only form of results for this lab are the command-line results, the only way to tell if there is an error in the program is through its results. The code correctly executes the tasks and prints the results with no errors, therefore it is logical to say that there is zero experimental error.

Conclusion

This lab left students much more comfortable using the debugging program gdb, which will be helpful for all future programming, as there are always errors. It also helped with comfortability using linked lists and structs in general. These concepts will be important leading up into classes, which are nearly identical to structs.

Appendix

Full contents of personList.cpp

```
#include <iostream>
#include <string>
using namespace std;

//Jack Fenton and Michael Delaney
//Northeastern University
//EECE2150 Embedded Design
//Lab 2
//30 January 2020

// Linked List Management Code
struct Person
{
   // Unique identifier for the person
   int id;
   // Information about person
   string name;
   int age;
   // Pointer to next person in list
   Person *next;
};
struct List
```

```
{
   // First person in the list. A value equal to NULL indicates that the
   // list is empty.
   Person *head;
   // Current person in the list. A value equal to NULL indicates a
   // past-the-end position.
   Person *current;
   // Pointer to the element appearing before 'current'. It can be NULL if
   // 'current' is NULL, or if 'current' is the first element in the list.
   Person *previous;
   // Number of persons in the list
   int count;
};

//Initialize Functions
void ListInitialize(List *list);
void ListNext(List *list);
void ListHead(List *list);
Person *ListGet(List *list);
void ListFind(List *list, int id);
void ListInsert(List *list, Person *person);
void ListRemove(List *list);
void PrintPerson(Person *person);

//User Functions
void AddPerson(List *list);
void FindPerson(List *list);
void RemovePerson(List *list);
void PrintList(List *list);
int inInt(string prompt);
string inString(string prompt);



/* main function: Will create and process a linked list */
int main() {
   List list;              // Create the main list
```

```
ListInitialize(&list);  // Initialize the list
int cur_case;
while(cur_case!=5)
{
    cout << endl <<
            "Main Menu \n\n" <<
            "1. Add a person \n" <<
            "2. Find a person \n" <<
            "3. Remove a person \n" <<
            "4. Print the list \n" <<
            "5. Exit \n\n" <<
            "Select and option: ";
        cin >> cur_case;
        if (cin.fail())
        {
                cin.clear();
                cin.ignore();
        }
    switch(cur_case)
    {
        case 1:
        //adding person
            cout << "Add a person \n\n";
            AddPerson(&list);
            break;

        case 2:
        //find person
            cout << "Find a person \n\n";
            FindPerson(&list);
            break;

        case 3:
        //remove person
            cout << "Remove a person \n\n";
            RemovePerson(&list);
            break;
```

```
            case 4:
            //print list
                cout << "Print the list \n\n";
                PrintList(&list);
                break;

            case 5:
            //exit
                cout << "Exit \n\n";
        }
    }
    return 0;
}
//end main


// Give an initial value to all the fields in the list
void ListInitialize(List *list)
{
    list->head = NULL;
    list->current = NULL;
    list->previous = NULL;
    list->count = 0;
}

// Move the current position in the list one element forward. If last element
// is exceeded, the current position is set to a special past-the-end value
void ListNext(List *list)
{
    if (list->current)
    {
        list->previous = list->current;
        list->current = list->current->next;
    }
}
```

```
// Move the current position to the first element in the list.
void ListHead(List *list)
{
   list->previous = NULL;
   list->current = list->head;
}

// Get the element at the current position, or NULL if the current position is
// past-the-end
Person *ListGet(List *list)
{
   return list->current;
}

// Set the current position to the person with the given id. If no person
// exists with that id, the current position is set to past-the-end
void ListFind(List *list, int id)
{
   ListHead(list);
   while (list->current && list->current->id != id)
       ListNext(list);
}

// Insert a person before the element at the current position in the list. If
// the current position is past-the-end, the person is inserted at the end of
// the list. The new person is made the new current element in the list
void ListInsert(List *list, Person *person)
{
   // Set 'next' pointer of current element
   person->next = list->current;
   // Set 'next' pointer of previous element. Treat the special case where
   // the current element was the head of the list
   if (list->current == list->head)
       list->head = person;
   else
       list->previous->next = person;
   // Set the current element to the new person
```

```
    list->current = person;
}

// Remove the current element in the list. The new current element will be the
// element that appeared right after the removed element
void ListRemove(List *list)
{
    // Ignore if current element is past-the-end
    if (!list->current)
        return;
    // Remove element. Consider special case where the current element is
    // in the head of the list
    if (list->current == list->head)
        list->head = list->current->next;
    else
        list->previous->next = list->current->next;
    // Free element, but save pointer to next element first
    Person *next = list->current->next;
    delete list->current;
    // Set new current element
    list->current = next;
}

void PrintPerson(Person *person)
{
    cout << "Person with ID: " << person->id << endl;
    cout << "\t  Name: " << person->name << endl;
    cout << "\t   Age: " << person->age << endl << endl;;
}

/*  Prints a prompt and takes input, returns input when it is int */
int inInt(string prompt)
{
    int inputInt;
    while(true)
    {
        cout << prompt;
```

```
        cin >> inputInt;
        if (cin.fail())
        {
            cin.clear();
            cin.ignore();
        }
        else break;
    }
    return inputInt;
}

/*  Prints a prompt and takes input, returns input as a string */
string inString(string prompt)
{
    string inputStr;
    cout << prompt;
    cin >> inputStr;
    return inputStr;
}



/*
Creates new newGuy, an empty struct with attributes of person
Assigns id, assigns name and age based on user input
From wherever the current position was, moves forward until position is one after last
defined person
Assigns last previously defined person to point to newGuy
*/
void AddPerson(List *list)
{
    Person *newGuy;
    newGuy= new Person;

    newGuy->id=list->count+1;
    newGuy->name=inString("Person's Name: ");
    newGuy->age=inInt("Person's Age: ");
    cout << endl;
```

```
   while (list->current!=NULL) ListNext(list);
   ListInsert(list, newGuy);
   list->count++;



}
/* New printing function that prints all data for one person on single line */
void PrintList(List *list)
{
   ListHead(list);
   while(list->current!=NULL)
   {
   cout << "id: "<< list->current->id << " Name: " << list->current->name << "  Age: "
<<list->current->age << endl;
   ListNext(list);
   }
}

/*
User declares id of person they want information for
ListFind() iterates through until current id is the one requested
PrintPerson() outputs information about person
*/
void FindPerson(List *list)
{
   int findID;
   do
   {
   findID=inInt("id for information: ");
   if (findID<=list->count) break;
   else cout << "This is not a valid id \n";
   } while(true);

   ListFind(list, findID);
   PrintPerson(list->current);
```

```
}
/*
Same way as FindPerson(), sets current position to the user input
Use predefined function to remove specified person
Reassigns previous element to point to one that follows
*/
void RemovePerson(List *list)
{
   int findID;
   do
   {
   findID=inInt("id to remove: ");
   if (findID<=list->count) break;
   else cout << "This is not a valid id \n";
   } while(true);

   ListFind(list, findID);
   ListRemove(list);

}
```