Lab Assignment 2
Linked Lists and the *gdb* Debugger

Saul Reyna, Yuheng Dong
reyna.s@husky.neu.edu
dong.yuh@husky.neu.edu

Submit date: 2/5/2020
Due Date: 2/5/2020

Abstract

The lab consisted of the use of a debugger that allows the user to go into what the program is doing step by step and the state of different variables and the memory, line by line. Upon compiling a C++ program, a flag -g is passed in the g++ command to allow the program being run through the debugger. Aside from letting the user run the program line by line, it allows the user to set a breakpoint that indicates the line to begin the step-by-step execution.

Introduction

Using the concept of linked list from the previous lab, a debugger can be used to allow the user to go through the behavior of the linked list and any actions performed to the linked list. This linked list is able to add an instance of the class Person, find a Person given an ID, remove a Person given an ID, and print all the Persons in the given list. Once the program is compiled, the debugger is run to look into the different states of the linked list at different locations of the program.

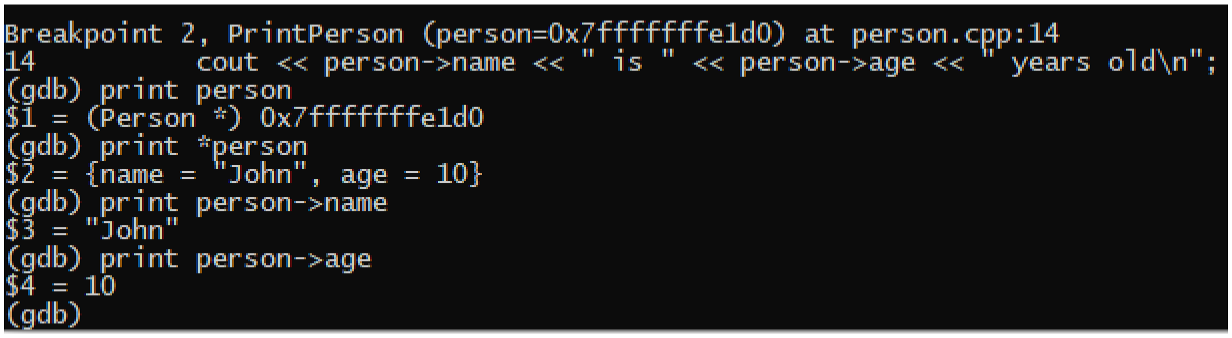
Lab Discussion

Equipment used in this lab include

- Zedboard (ZYNQ SoC)
    - Dual ARM Cortex – A9 MPCore with CoreSight
    - Zynq-7000 AP SoC XC7Z020-1CLG484
- Xillinux OS
- CoE lab desktop

Results and Analysis

Assignment 1



```
Breakpoint 2, PrintPerson (person=0x7fffffffe1d0) at person.cpp:14
14          cout << person->name << " is " << person->age << " years old\n";
(gdb) print person
$1 = (Person *) 0x7fffffffe1d0
(gdb) print *person
$2 = {name = "John", age = 10}
(gdb) print person->name
$3 = "John"
(gdb) print person->age
$4 = 10
(gdb)
```

Before the program stops at the breakpoint, the instance of Person is instantiated, so the members for Person have been assigned. Once the debugger reaches the breakpoint, different elements of the instance are printed. The first print is printing the actual instance of the Person, which prints out the address of the pointer that leads to the Person. The next print prints the actual contents of the aforementioned address, which would be all the assigned members of the Person at said address. The next print command references the “name” member of the Person and prints it out, in this case, “John”. Similarly, the final print references the “age” member of the Person and prints it out in the same fashion as the previous print command.

Assignment 1.2

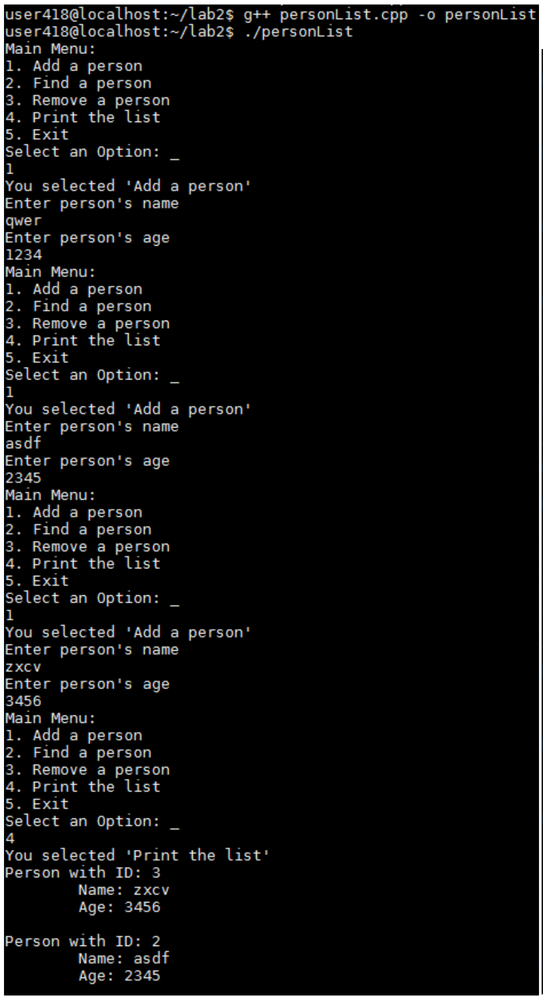
```
#include <iostream>
#include <string>

using std::cout;
using std::cin;
using std::endl;
using std::string;

int counter=1;

// Linked List Management Code
struct Person
{
    //s 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;
};

// 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 << "\tName: " << person->name << endl;
    cout << "\tAge: " << person->age << endl << endl;
}
void AddPerson(List *pist)
{
    int age;
    string name;
    cout<<"Enter person's name"<<endl;
    cin>>name;
    cout<<"Enter person's age"<<endl;
    cin>>age;
    pist->count+=1;
    Person *person = new Person;
    person->name=name;
    person->age=age;
    person->id=counter;
    counter++;
    ListInsert(pist, person);
```

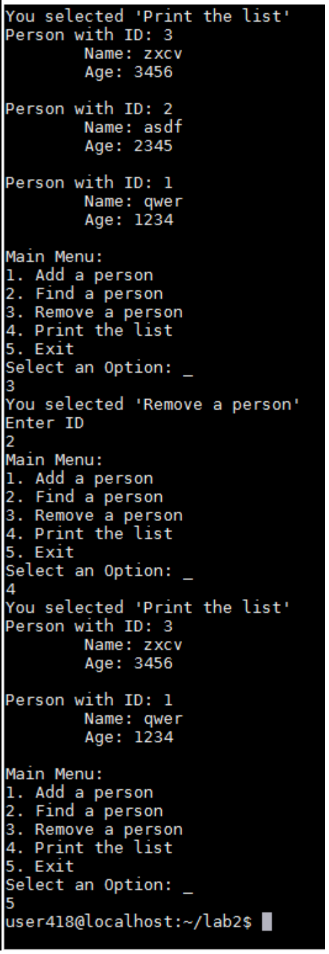
```
}
void FindPerson(List *pist)
{
    int id;
    cout<<"Enter ID"<<endl;
    cin>>id;
    ListFind(pist, id);
    if(id>pist->count)
    {
        cout<<"There is no person with that id"<<endl;
    }
    else
    {
         PrintPerson(ListGet(pist));
    }
}
void RemovePerson(List *pist)
{
    int id;
    cout<<"Enter ID"<<endl;
    cin>>id;
    ListFind(pist, id);
    if(id>pist->count)
    {
        cout<<"There is no person with that id"<<endl;
    }
    else
    {
        ListRemove(pist);
        pist->count--;
    }
}
void PrintPist(List *pist)
{
    ListHead(pist);
    for(int i = 0;i<pist->count;i++)
    {
        PrintPerson(NULL);
        ListNext(pist);
    }

}
/** main function: Will create and process a linked list
 */
```

```
int main() {
    List list;                  // Create the main list
    ListInitialize(&list);          // Initialize the list
//*************** PUT THE REST OF YOUR CODE HERE  *****************
    int input;
    while (true){
        cout << "Main Menu:" << endl;
        cout << "1. Add a person" << endl;
        cout << "2. Find a person" << endl;
        cout << "3. Remove a person" << endl;
        cout << "4. Print the list" << endl;
        cout << "5. Exit" << endl;
        cout << "Select an Option: _ " << endl;
        cin >> input;
        switch (input){
            case 1:
                cout << "You selected 'Add a person'" << endl;
                AddPerson(&list);
                break;
            case 2:
                cout << "You selected 'Find a person'" << endl;
                FindPerson(&list);
                break;
            case 3:
                cout << "You selected 'Remove a person'" << endl;
                RemovePerson(&list);
                break;
            case 4:
                cout << "You selected 'Print the list'" << endl;
                PrintPist(&list);
                break;
            case 5:
                return(0);
                break;
            default:
                cout<<"Error: Invalid number"<<endl;
                return(0);
                break;
        }
    }
}
```

Assignment 1.3



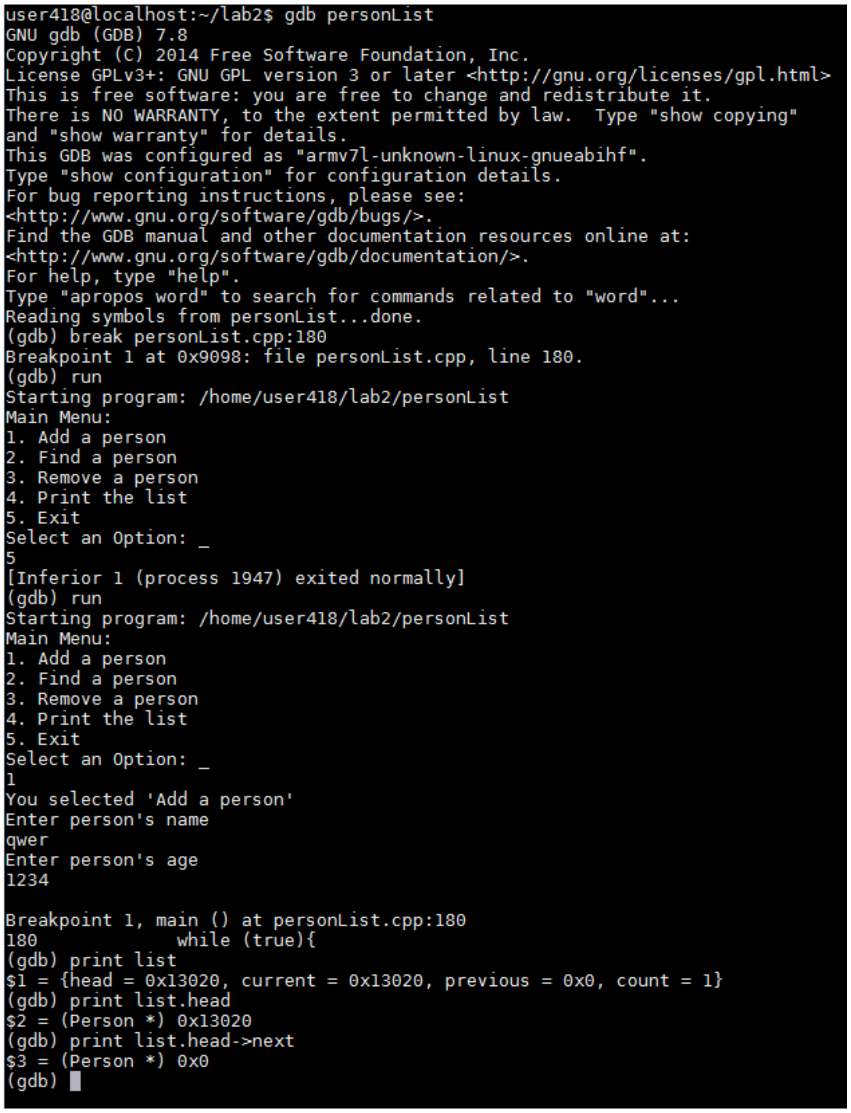
```
user418@localhost:~/lab2$ g++ personList.cpp -o personList
user418@localhost:~/lab2$ ./personList
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
1
You selected 'Add a person'
Enter person's name
qwer
Enter person's age
1234
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
1
You selected 'Add a person'
Enter person's name
asdf
Enter person's age
2345
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
1
You selected 'Add a person'
Enter person's name
zxcv
Enter person's age
3456
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
4
You selected 'Print the list'
Person with ID: 3
        Name: zxcv
        Age: 3456

Person with ID: 2
        Name: asdf
        Age: 2345
```



```
You selected 'Print the list'
Person with ID: 3
        Name: zxcv
        Age: 3456

Person with ID: 2
        Name: asdf
        Age: 2345

Person with ID: 1
        Name: qwer
        Age: 1234

Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
3
You selected 'Remove a person'
Enter ID
2
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
4
You selected 'Print the list'
Person with ID: 3
        Name: zxcv
        Age: 3456

Person with ID: 1
        Name: qwer
        Age: 1234

Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
5
user418@localhost:~/lab2$ 
```

The first option chosen was to add a person to the list with the name “qwer” and an age of 1234. Two more persons were added to the list and then the list was printed. Both of these actions print the expected results since each of the persons have the respective IDs, relative to the time that person added to the list (i.e. the first person added should have an ID of 1, the next person an ID of 2, and so on). After printing out the list of people, the person with ID 2 was removed from the list. The print result was also within the expected results, since removing that person should leave the person with ID’s 1 and 3 intact. Upon removing the person and printing the updated list, the print results matched this behavior.

Assignment 1.4



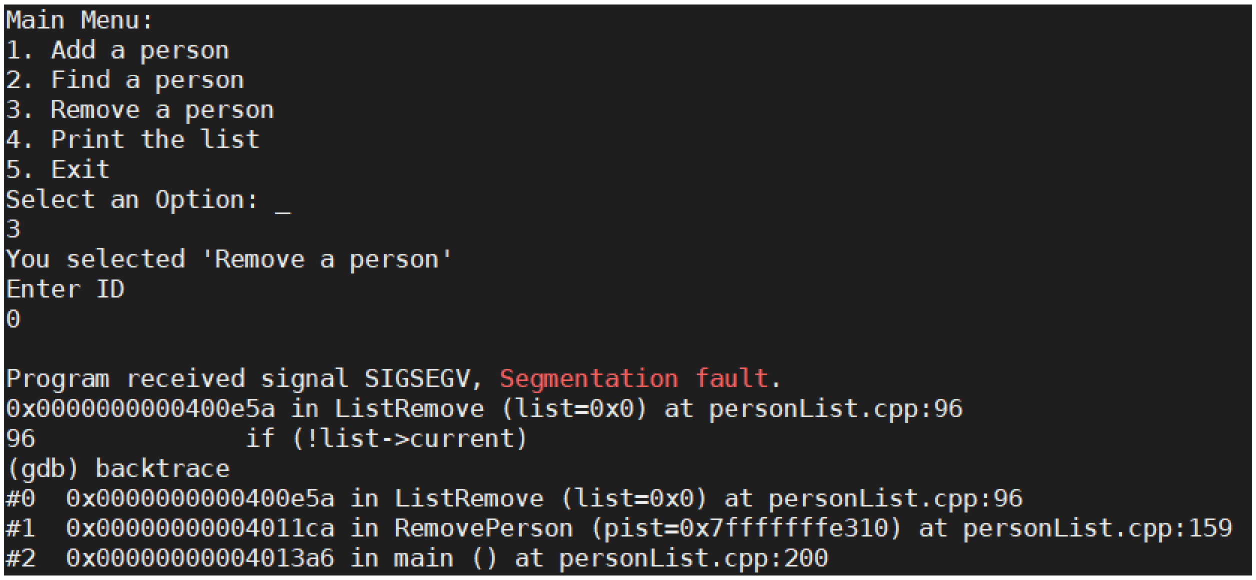
```
user418@localhost:~/lab2$ gdb personList
GNU gdb (GDB) 7.8
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "armv7l-unknown-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from personList...done.
(gdb) break personList.cpp:180
Breakpoint 1 at 0x9098: file personList.cpp, line 180.
(gdb) run
Starting program: /home/user418/lab2/personList
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
5
[Inferior 1 (process 1947) exited normally]
(gdb) run
Starting program: /home/user418/lab2/personList
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
1
You selected 'Add a person'
Enter person's name
qwer
Enter person's age
1234

Breakpoint 1, main () at personList.cpp:180
180             while (true){
(gdb) print list
$1 = {head = 0x13020, current = 0x13020, previous = 0x0, count = 1}
(gdb) print list.head
$2 = (Person *) 0x13020
(gdb) print list.head->next
$3 = (Person *) 0x0
(gdb) 
```

Figure 1 . GDB Output

The first command run was break personList.cpp:180, which sets a breakpoint at line 180 (where before the main menu is printed for the second time).

The next command was simply to run the program. As per the assignment instructions, option 1 was selected and a random person with a random age was added to the list. The breakpoint that was set by the first command is then reached as the main menu is about to be displayed again. The third command is print list, which printed the contents of the list and their addresses, as well as the current count. Print list.head print the address of the head of the list (0x13020). Finally, print list.head->next moved the head address to the next address 0x0.

Assignment 1.5 – Shrinking the vector



```
Main Menu:
1. Add a person
2. Find a person
3. Remove a person
4. Print the list
5. Exit
Select an Option: _
3
You selected 'Remove a person'
Enter ID
0

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400e5a in ListRemove (list=0x0) at personList.cpp:96
96              if (!list->current)
(gdb) backtrace
#0  0x0000000000400e5a in ListRemove (list=0x0) at personList.cpp:96
#1  0x00000000004011ca in RemovePerson (pist=0x7fffffffe310) at personList.cpp:159
#2  0x00000000004013a6 in main () at personList.cpp:200
```

Figure 2 . GDB output with intentionally broken code

After compiling the program with debug flags, we then ran the program in gdb. In this instance, an input for ListRemove was replaced with NULL. We then used backtrace to get the value that is causing the issue (list=0x0, the NULL value). It also gives the address and values of where ListRemove is later called in RemovePerson and the main function.

Conclusion

The use of a debugger allows to break down code when a program becomes unwieldy for a person to follow through. When it becomes unwieldy, some action might be overlooked or having to keep track of every variable becomes too much of a task, so the debugger allows the programmer to break it down to pieces to where the programmer might think it might fail or wherever any of the errors that the program outputs tells the programmer that there is a problem. This gives the ability to correct any mistakes that arose while writing the code. Similar to having someone proofread your essay before turning it to check for any mistakes that the author might have overlooked due to the human nature, the programmer has a computer to guide them and show them were the mistakes were made while also giving them a hint on how to fix the mistake by telling them why exactly the program failed.

References

[1] Michael Benjamin, “*Lab Report Guide*”, Northeastern University, January 18 2006.