Lab Assignment 2
Linked Lists and the *gdb* Debugger

Lab 2.0 Introduction

In this lab we will practice the use of *gdb* as a tool to debug your programs through step-by-step execution and memory inspection. We will also continue to work with linked lists as an alternative data structure to store sequence of elements, where insertions and deletions have a constant cost. Finally, we will use *gdb* to explore the execution of a main program using linked lists.

Lab 2.1 The *gdb* debugger

A debugger is a tool used to inspect the memory of a program in a controlled execution environment, with the common objective of identifying the presence of bugs in the program. The *gdb* (GNU debugger) tool is shipped together with *gcc* in most Linux distributions, and can be accessed both on the ZedBoard and the COE machines. Consider the following program, which assigns predefined values to a variable of type Person, and prints them through an invocation to function PrintPerson:

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

struct Person
{
     string name;
     int age;
};

void PrintPerson(Person *person)
{
     cout << person->name << " is " << person->age << " years old\n";
}

int main()
{
     Person person;

     person.name = "John";
     person.age = 10;

     PrintPerson(&person);
}
```

In order to debug this program with *gdb*, you need to make sure that it was compiled with additional debug information, required by *gdb* to associate instructions in the binary file with lines of your source code. This is done by adding flag -g to your *g++* command line:

```
$ g++ person.cpp -o person -g
```

When you first run *gdb*, all you see is a command prompt, in a similar way as when you run a UNIX shell. This is where you can enter *gdb* commands that will let you run your program under the debugger's supervision. You can run executable file person on *gdb* with the following commands:

```
$ gdb

(gdb) file person                          ← Loads file “person”
Reading symbols from “person”... done

(gdb) run                              ← Runs it
Starting program “person”...
John is 10 years old
[Exited normally]

(gdb) quit                                 ← Exits gdb
```

Lab 2.2 Step-by-step execution

With *gdb* you can control the pace at which your program executes by advancing its state only by one line of code at a time. You can use *gdb* commands start and next to run function main() in a line-by-line basis. Notice that the example below loads program person by passing it as an argument to *gdb*, instead of using command file person; these are two equivalent methods.

```
$ gdb person

(gdb) start
Starting program: person
18   {

(gdb) next
21        person.name = "John";

(gdb) next
22        person.age = 10;

(gdb) next
24        PrintPerson(&person);

(gdb) next
John is 10 years old
```

```
(gdb) next
26   };

(gdb) next
[Inferior 1 (process 22747) exited normally]
(gdb) quit
```

You can observe in the output that *gdb* displays the line of source code that will execute upon the next call to command next. When the line of source code shows the closing brace of function main(), running command next again will finish the program, as reported by *gdb* in the last line of output above.

You can run the same *gdb* command over and over by simply pressing *Enter* repeatedly after having entered the command once. Pressing *Enter* with an empty command will run the previously executed *gdb* command.

Lab 2.3 Examining the program memory

At any time during the execution of your program, you can inspect the current value of the program variables using the print command in *gdb*. This command takes a C++ expression as an argument, which can include variable names combined with optional useful operators, such as dereference, array access, or arithmetic.

```
$ gdb person

(gdb) start
Starting program: person
18   {

(gdb) next
21        person.name = "John";

(gdb) next
22        person.age = 10;

(gdb) print person.name
$1 = "John"

(gdb) print person.name[0]
$2 = (const char &) @0x603028: 74 'J'
```

Pre-lab assignment - Part I:

Prepare a copy of program `person.cpp` and compile it to produce executable file person, with the appropriate debug flags. Run your program step by step using command next until the moment you reach the invocation to function `PrintPerson`. At this point, run command step instead, and then continue running next until your program finishes. Use at least two print commands that show the values of program variables at any time during the execution.

Screen capture the complete *gdb* execution session, and explain each *gdb* command used. Describe the outputs observed from running those commands.

Pre-lab assignment - Part II:

a) Read carefully the section “Linked list management” on page 7 of this document.

b) Download and study the provided startup linked list management C++ program (`personList.cpp`).

c) In the `main` function of the provided C++ program, and using the COE Linux machines, write a program (or copy and modify your code from the previous lab) that displays the menu shown on section 2.6 below, and waits for the user to enter an option. If the option is invalid, an error message should be displayed and the main menu should be shown again. When the user selects option 5, the program should finish. When the user enters any other valid option, your pre-lab program should just print the name of the option on the screen. We will replace the code for each option with its actual functionality in the lab assignments. For example:

```
Select an option: 4
You selected “Print the list”
```

Submit your pre-lab code and sample output in your PDF file on Blackboard

Lab 2.4 Breakpoints

Running an entire program step by step can be a tedious task, especially for larger programs, if all you want is inspect the value of a variable at an advanced point of the execution. This is solved by inserting a breakpoint at a specific line of code in your program, and letting the program run without interruption until that line is reached. A breakpoint can be added with command break, followed by a file name and line number, or by a function name. After a breakpoint has been reached, you can resume the execution of your program using command continue, or you can return to a step-by-step execution with commands next and step.

```
$ gdb person

(gdb) break person.cpp:11                ← (also “break PrintPerson”)
Breakpoint 1 at 0x4005a9: file person.c, line 11c.

(gdb) run
Breakpoint 1, PrintPerson (person=0x7fffffffe4b0) at person.c:13
13       cout << person->name << “ is “ << person->age << “ years old\n”;

(gdb) continue
Continuing.
John is 10 years old
[Inferior 1 (process 22807) exited normally]
```

Assignment 1

Load program *person* on *gdb* and set a breakpoint at the beginning of function PrintPerson. Run your program without interruption up to this point, and the run the following *gdb* commands:

```
(gdb) print person
(gdb) print *person
(gdb) print person->name
(gdb) print person->age
```

Provide the output given by *gdb* for all these commands, and explain it in detail.

Lab 2.5 The stack trace

When your program is stopped, either through a step-by-step execution or at a breakpoint, *gdb* lets you explore the sequence of function calls performed since function main() in order to reach that point of the execution of the program, using command backtrace. More importantly, *gdb* lets you navigate through the so-called *call stack*, in order to explore the current value of local variables in any of the function calls made to reach the current point.

The following example makes *gdb* stop at the moment where function PrintPerson has been reached, and shows the current state of the call stack. Then it navigates through the call stack with commands up and down, and prints variable person in both the context of function main() and the context of function PrintPerson(). Although these two identifiers share the same name, they refer to different variables. As a matter of fact, they even have different types (Person vs. Person *).

```
$ gdb person

(gdb) break person.cpp:11
Breakpoint 1 at 0x4005a9: file person.c, line 11.

(gdb) backtrace
#0  PrintPerson (person=0x7fffffffe4b0) at person.c:12
#1  0x0000000000400600 in main () at person.c:24

(gdb) up
#1  0x0000000000400600 in main () at person.c:24
23        PrintPerson(&person);

(gdb) print person
$1 = {
  name = "John",
  age = 10}

(gdb) down
#0  PrintPerson (person=0x7fffffffe4b0) at person.c:13
13        cout << person->name << " is " << person->age << " years old\n";

(gdb) print person
$2 = (struct Person *) 0x7fffffffe4b0
```

Lab 2.6 Linked list management

We have seen in class how to use linked lists to efficiently insert and remove elements from a sequence in constant time. In the additional online material for the course, you can find the code for the data structure definitions required to implement a linked list of persons, as well as the functions needed to manipulate the list.

We will start with the implementation of a main program to test the linked list, with a similar structure to the one used to test the dynamically growing array in the previous lab. It should display a menu with the following options:

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

- When the user selects option 1, the program should ask the user to enter information for a new person, including the person's name and age. Notice that the Person data structure we're working with now has an additional field id. This field should be assigned automatically by your program in increasing order; that is, the person created first should have an ID equal to 1, the second person should have an ID equal to 2, etc. Use function ListInsert() as needed.
- When the user selects option 2, the program should ask the user to enter a person's ID. If a person with that ID is found, the program should print the person's full profile (ID, name, and age). If no person with that ID exists, a proper error message should be displayed. Use functions ListFind(), ListGet(), and PrintPerson() as needed.
- When the user selects option 3, the program should ask the user to enter a person's ID. If a person with that ID is found, it should be removed from the list. If no person exists with that ID, a proper error message should be displayed. Use functions ListFind(), ListGet(), and ListRemove() as needed.
- When the user selects option 4, the content of the entire list should be printed. Use the list traversal functions ListHead() and ListNext() as needed, as well as functions ListGet() and PrintPerson().

If any option other than 5 was selected, the main menu should be displayed again, requesting another action from the user. This process will repeat until option 5 (exit) is selected.

Assignment 2

Write a program that includes all type definitions and functions presented in class to manage the List and Person data structures, together with an implementation of the main program complying with the specification above. Show the complete code of your program.

Assignment 3

Run your program several times in order to test the behavior of each menu option. For each option, show the output of your program and explain whether you are obtaining the expected results.

Assignment 4

Use *gdb* to run the linked list main program developed in the first part of this lab assignment, and set a breakpoint at the end of the loop in function main(), right before the main menu is printed for the second time.

Run your program and choose the menu option that allows you to insert a new element in the list. Once the program stops at the given breakpoint, run the following commands (assuming that list is the name you gave to the variable representing your linked list):

```
(gdb) print list;
(gdb) print list.head;
(gdb) print list.head->next
```

Provide the full sequence of commands you ran on *gdb* to reach this point, as well as the output of these commands. Explain the output provided by *gdb*.

Lab 2.7 Debugging program crashes

Modify the call to function PrintPerson() in the main program with the following line of code:

```
PrintPerson(nullptr);
```

When you compile and run program *person*, it will crash right away with the following message:

```
Segmentation fault (core dumped)
```

A segmentation fault occurs when your program is accessing an invalid memory location, that is, one that was not associated with any valid variable either allocated statically by the compiler, or dynamically through the operator *new*. In general, it is hard to deduce where the problem in your code could be by just observing this error message. This is where *gdb* can be very useful.

If we run the buggy version of our program on *gdb*, the following sequence of commands would reveal that the reason why it crashed is that we are trying to dereference a null pointer in the cout statement of function PrintPerson():

```
$ gdb person

(gdb) run
Starting program: person
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005b5 in PrintPerson (person=0x0) at person.c:13
13        printf("%s is %d years old\n",

(gdb) backtrace
#0  0x00000000004005b5 in PrintPerson (person=0x0) at person.c:13
#1  0x0000000000400608 in main () at person.c:25

(gdb) print person
$1 = (struct Person *) 0x0
```

Assignment 5

Modify any position of the code in your linked list program strategically in order to make it perform an invalid memory operation that makes the program crash—alternatively, and more desirably, use a real intermediate unstable version of your own code that produced a program crash during the development of previous assignments.

Run the program on *gdb* and show the sequence of *gdb* commands you would run to infer the value of the variable that is causing the problem. Show the full output of *gdb* for your commands and explain it.

Extra credit

Add an additional option to the menu in the main program that allows you to sort the list of persons. When this option is selected, the program should ask the user whether the list of persons should be sorted by age, or by name.

A possible strategy to sort a linked list is creating a temporary array with the same elements, sorting the elements on the array, and reconstructing the linked list based on the sorted array. Feel free to use any alternative method, and make sure you describe it accurately. When sorting persons by name, you can use the alphabetical order of strings.

Lab Report

No formal lab report required for this lab. Submit all your responses (commented code and sample output) from the different parts in a single PDF file (one per team).