Lab 5 - Debugging Revisited
CPSC 1021 - Fall 2016

Due: Wednesday, September 21 2016, 11:59:59 Midnight
Introduction
Throughout the semester, we will continue to see advanced features of the C programming language and introduce you to the concepts of object-oriented programming (OOP).

Since you have experience programming, you've probably discovered that as software projects get larger and more complicated, debugging becomes increasingly challenging. On your smaller projects, using the debugging technique of printing out variables probably was your preferred solution to finding bugs in your code. However, printing out every variable simply doesn't scale as projects get larger. If you were working on the iROAR programming team trying to isolate a bug in the registration process, you'd have a minimum of 17,000 students registering for multiple classes and labs. Printing out every variable would just give you screen after screen of information and bury you.

You may have been introduced to debugging using the ***gdb*** utility in your previous courses. This lab is a refresher on ***gdb*** and also introduces you to some utilities that we will be using throughout the course and lab: SCP, tar, and Valgrind

Lab Objectives
- Use SCP to obtain files from the lab directory and use tar to uncompress a tarball
- Use the manual pages to learn more about command line optional flags
- Use tar to create a single tarball containing your lab handins.
- Debug a C program using ***gdb*** (via breakpoints, checking the value of a variable during program execution, and tracing function calls)
- Use Valgrind to help you debug a program and understand memory management

Prior to Lab
- Review the ***gdb*** Cheat Sheet
  http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Lab Instructions

Part 1: Tips for Housekeeping Files: Tar and SCP

In some of your classes or labs, you may be asked to submit your files as a single, compressed file. On your typical Windows or Macintosh computer, this is commonly referred to as "zipping" your files. ZIP is a file format used to archive files that originated in 1989 by Phil Katz. Most UNIX and UNIX-like operating systems have a built in utility to archive files called tar (tape archive). It is a very common storage method for grouping up a set of a files to email or copy to another location.

How to use the tar utility:

1. Place all the files you want bundled together in the same directory. Let's assume you create a directory called **1021_Lab_5** that contains three **.c** code files.
2. From the parent directory, you would run the following command:

```
tar cvf I_Luv_Labs.tar 1021_Lab_5/
```

Where **c** indicates tar to create a new tar file, the **f** indicates tar to name it **I_Luv_Labs.tar**, and v (verbose) displays the files that are being compressed in the terminal window. The v is optional but very useful. Note that you have to supply the **.tar** file extension to the command.

Typically, the process of creating a .tar file is called creating a tarball or "tarballing" your files.

Lab Exercises.
Answer the following questions by experimenting with command line optional flags or by looking up information using the manual pages or check out this link http://lmgtfy.com/?q=how+to+create+a+tar.gz+file+from+the+command+line&l=1.

1. Give the full command to tarball up the files listed in a subdirectory called **1021_Lab_5** where you want to compress the tarball (e.g., reduce the amount of space needed for the tarball). This command is slightly different than the command listed above. It will result in your tarred file having a .tar.gz extension.

2. How would you "unzip" the tarball file you created in exercise 1?

3. What does the **.gz** file extension stand for?

In the next part of the lab, we will focus on a new way that can obtain files, using the Secure Copy (SCP) utility. SSH is the acronym for Secure Shell which allows you to create an encrypted remote connection using the command-line. SCP is a way that allows you to transfer files from a host to another host, also using encryption.

```
scp source_file_name username@destination_host:destination_folder
```

However, that basic command will copy files in the background and you will not see any indication of what is happening until the process is done or an error occurs. The **–v** flag is used to display debugging information onto the screen (think of the v as standing for verbose).

The following table gives some common SCP options

-p	Estimation of the time and connection speed
-C	Compresses your files for transport
-c	Allows you to change the encryption standard
-P	Specify the connection port to use (port 22 is typically the default)
-r	Recursively copy all files and subdirectories (and their files) within a directory

Create a folder in your account for this lab (e.g., 1020_Lab5). Change the current directory to the folder and use the following **scp** command to copy files to your computer.

```
cp /group/course/cpsc1020/f16_labs/lab5/* .
```

You'd want to use SCP to do this if you're currently off campus. To use SCP to accomplish the same task, you may run the following command (as one line, excuse the word wrapping):

```
scp yourusername@koala1.cs.clemson.edu:/group/course/cpsc1020/f16_labs/lab5/* .
```

"Unzip" the lab05files.tar tarball—you'll now have the files that you need for the next part of the lab.

Part 2: Dusting off your gdb debugging skills

Some helpful gdb tutorials:

https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf
https://www.tutorialspoint.com/gnu_debugger/index.htm
http://www.dirac.org/linux/gdb/
http://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html
http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/

Debugging is often the most frustrating part of programming. It's important to develop good debugging strategies early on to help you deal with undesirable program behavior in the future. If you have not already done so, please look at the tutorial found at this link: www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf

The **gdb** utility is a powerful program that can help you isolate bugs in computer programs. Your instructors can attest that many students will come in to office hours saying, "My program seg faults and I don't know where". Use **gdb** and it'll tell you! As a reminder, **gdb** allows you to:

- Start your program, specifying anything that might affect its behavior such as command-line arguments
- Make your program stop on specified conditions.
- Examine what has happened after your program has stopped running
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

First, we will work with **LinkedList.c** from the tarball you copied earlier in the lab. In a linked data structure, information is contained within a "node" and each node points to the next node on the list. Linked data structures are examples of dynamic data structures and can easily grow as you have more data to add (as long as the computer has available memory!) When an array fills up and you still have values to place into it, you must create a new, larger array and copy over your prior array—this can be very "costly" when you are dealing with many, many items.

For this part of the lab, compile and run **LinkedList.c**. You'll notice that the program crashes once you try to run it. Without looking at the source code:

- Recompile **LinkedList.c** using the **gdb** flag
- Run **LinkedList.c** in **gdb** and use **gdb** to isolate the error.
  - Hint: spend some time in **gdb** looking at the variables and figure out where you should set breakpoints.

Once you've isolated the error, create a copy of **LinkedList.c** called **LinkedListFixed.c** and fix the error within the code to allow the program to function properly.

Next, we will work with **Mystery.c**. Compile and run **Mystery.c**, noting that it takes in command line arguments. Play around with a few calls to the program to see if you can figure out how many command line parameters it takes. Make note when there are crashes and when output doesn't seem to correspond to what you think should be happening. Without looking at the source code:

- Recompile **Mystery.c** using the **gdb** flag
- Run **Mystery.c** in **gdb** and use **gdb** to isolate the error.
  - Hint: You'll also need to run command line arguments in **gdb**. If a program takes 5 command line arguments, you'd use the following **gdb** command: "run 1 2 3 4 5" where each number represents one of the command line arguments.

Once you've isolated the error, create a copy of **Mystery.c** called **MysteryFixed.c** and fix the error within the code to allow the program to function properly.

Part 3: Valgrind debugging utility

Valgrind is a debugging tool to help you understand memory management and threading bugs. (Threading refers to multiple processes of execution—we haven't covered that yet but you will learn about that later on in your undergraduate careers). First, however, are two cautions about Valgrind:

1. it isn't perfect -- it won't catch everything,
2. it is SLOWWWWW. Your program may run 10-20 times slower under Valgrind

You don't need to run your program every time under Valgrind. You can use it periodically during the debugging stage for the program but discontinue its use when things have stabilized.

```
/* memoryLeak.c
example to use for valgrind

this program will reserve a block of memory
large enough to hold 100 characters */
#include <stdlib.h>

int main(void) {
        char *x = malloc(sizeof(char) * 100);
        return 0;
}
```

1. Compile the program using `gcc -g memoryLeak.c -o memoryLeak`
2. Run Valgrind via `valgrind --tool=memcheck --leak-check=yes ./memoryLeak`
3. View the following:

```
==16204== Memcheck, a memory error detector
==16204==  Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al
==16204== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==16204== Command: memoryLeak
==16204==
==16204==
==16204== HEAP SUMMARY:
==16204== in use at exit: 100 bytes in 1 blocks
==16204== total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==16204==
==16204== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16204==
==16204==  at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16204== by 0x400505: main (memoryLeak.c:11)
==16204==
==16204== LEAK SUMMARY:
==16204== definitely lost: 100 bytes in 1 blocks
==16204== indirectly lost: 0 bytes in 0 blocks
==16204== possibly lost: 0 bytes in 0 blocks
==16204== still reachable: 0 bytes in 0 blocks
==16204== suppressed: 0 bytes in 0 blocks
==16204==
==16204== For counts of detected and suppressed errors, rerun with: -v
==16204==ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

The highlighted lines in the center shows how much memory is lost and where in the program the lost memory was allocated.

There will be times when the **--leak-check=yes** option will not result in showing you all memory leaks. To find absolutely every unpaired call to free or new, you'll need to use the **--show-reachable=yes** option. Its output is almost exactly the same, but it will show more unfreed memory.

Compile bad.c and then run it using valgrind, see example above. Use the output from valgrind to determine the problem with this file. Hint: There are two main issues with this file.
Correct the problems, run valgrind again to make sure you corrected all problems. Using comments, explain the problem with the program. Rename the corrected file BadFixed.c

Additional Practice

- Valgrind tutorial: http://www.cprogramming.com/debugging/valgrind.html

Submission Instructions

- Use handin (**http://handin.cs.clemson.edu)** to submit a tarball (tar.gz file) containing your **LinkedListFixed.c, BadFixed.c, MysteryFixed.c**, and a file containing your answers to Exercises 1, 2, 3. The file should be labeled **username.txt** where username is your username. Include the appropriate header comments to all code files submitted.
- If you are unable to submit your lab via handin by the due date, email your lab instructor before the due date with any questions or difficulties.