Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Lab 6, March 2/3

In this lab, you will first use `gdb` to debug an incorrect program. After this you will learn how to generate random numbers using C library functions, and use them to solve some problems.

Be sure to get help from teaching assistants whenever you have any questions.

1. First, perform the following steps to get started:

   (i) Login to server bluenose.cs.dal.ca via SSH from a CS Teaching Lab computer or from your own computer.

   (ii) Change your current working directory to the `csci2132` directory created in Lab 1.

   (iii) Create a subdirectory named lab6.

   (iv) Change your current working directory to this new directory.

   (v) Copy the file `/users/faculty/prof2132/public/lab6/binary.c` to the directory you just created.

2. The `binary.c` file that you copied over in Question 1 is supposed to be a program that does binary search. It first asks the user to enter a positive integer. It then issues an error message if the user does not correctly input a positive integer. Next it calls the function `binary_search` to perform binary search in an array of size 10, whose values are initialized when the array is declared. If the integer is found in this array, this program prints the location in this array at which this integer is stored. Here the location starts at 1 and ends at 10 (be careful of the array subscripts). Otherwise, this program prints a message to say that the integer is not found.

   There are, however, two bugs in this program. In this lab, do not try to locate the errors by reading the code. Instead, follow the instructions to use `gdb` to find the bugs. The debugging skills are very important: For long programs, this is often the only sensible way of finding bugs.

   Now, use the following command which you can invoke inside emacs to compile this program to make it ready for debugging, and name the executable file `binary`:

   ```
   gcc -g -o binary binary.c
   ```

In Question 3, we will locate and fix the first bug, while in Question 4, we will address the second. You may find the list command of gdb helpful as it can be used to check the source code. Type `help list` in your gdb console to find out how to use this command.

3. Enter the command `./binary` to run it. When it prompts for user input, enter 35. What do you see?

   Now, let's fix the bug that you saw by following the instructions below (you can use abbreviations of the commands, e.g. `p array` is the same as `print array`):

   (a) Start gdb using `gdb binary`.

   (b) In the gdb console, enter `break main`. This will set a break point at the main function.

   (c) Enter `run` to run this program using gdb.

   (d) When the program pauses, enter `step` once.

   (e) Enter `print array` to print the content of `array`. So far everything seems to be good.

   (f) Enter `step` twice. The program now prompts for user input.

   (g) Enter 35 as your input.

   (h) Enter `print key`.

   You can now see that the user input is correctly assigned to the variable key. However, if you enter `step` again, you will see that the following statement is executed:

   ```
   printf("Please enter a positive integer.\n");
   ```

   This is incorrect as we know that 35 is a positive integer. Thus, this is the first place in which the program state does not match the expected state, and we can conclude that the if-test must be incorrect. Thus, check line 13 in the source file to find the error and fix it. You can use the gdb command `kill` to stop the program currently being debugged. You can then quit `gdb` so that you can modify the source code, or open another terminal to edit the code.

4. After you fix the bug in the if-test, compile and run this program again, and use 35 as the input. The program prints `35 is at location 5.` However, by checking line 9 of this program, we can see that 35 is not stored in the array at all. This means that our program is still buggy.

   In Question 3, we started debugging from the beginning of this program. This means that we used the linear approach as described in Lecture 14 (see the slides given online). This may take too much time if the program is very long. Now we use a different strategy. We will pause at the function `binary_search` (we will learn how to define a function later this term; for now, you can treat a C function as a Java

method, even though it is not a member of any class), and check if its parameters are all assigned correct values. If they are, then the bug occurs after the values of arguments are passed, and we keep stepping through the program. Otherwise, we will check the part of the program before the function call.

Follow the steps below:

(a) Enter `delete 1` to remove the first breakpoint that we set (it happened to be the only breakpoint that we set for Question 3). Note: this assumes that you never exited the gdb program that you invoked. If you start over, then this is not necessary. We also assume that you used the kill command already as suggested in Question 3.

(b) Enter `break binary_search` to set a breakpoint.

(c) Enter `run` to run this program again using gdb. Use 35 as the user input again.

(d) Enter `print *array@10` to print the 10 elements of the parameter array. Using `print array` will however shows its memory address instead. This applies to array parameters. When you learn pointers, it will be easier for you to see why it works this way; for now, just use this command.

(e) Enter `print len` and `print key`. Now you can see that all the parameters of this function have correct values. Therefore, the bug must occur after the breakpoint.

(f) Enter these commands: `display lower`, `display upper` and `display middle`. This way, whenever we enter the `step` command, the new values of these three variables will be automatically displayed. For now, the values of some of these variables are random numbers, but this is okay as they have not been assigned any values yet.

(g) Keep entering the `step` command until you finish executing the while loop. Each time you enter this command, verify the values of lower, upper and middle, to see if their values are different from their expected values. In this process, you will find that they always have the correct values. Therefore, the while loop is correct.

(h) Now enter `print array[middle]`. You will see 34. Therefore, `array[middle]` is not equal to `key`. However, if you enter the `step` command again, the if statement will branch into `return middle`, which is incorrect.

From the above, we can conclude that the bug is in line 45, which is the if-test. Read the code in this line to find the bug and fix it. Now, if you recompile and use 35 as the user input, the program will correctly print that this number is not found.

5. So far we have fixed these two bugs. It is time to thoroughly test this program. In addition to test some regular cases, also test the following boundary and error cases:

(a) Use the smallest number in the array as user input;

(b) Use the largest number in the array as user input;

(c) Use a positive integer that is less than the smallest value;

(d) Use a positive integer that is greater than the largest value;

(e) Use a negative value as user input;

(f) Use a character as user input.

6. Random number generators are very useful. For example, if you are writing a dice game to be played between players, you will need to generated a random number between 1 and 6 to simulate dice throws.

   Random number generators are also useful for implementing randomized algorithms.

   To generated random numbers, we use several C functions. The function `time` (from `time.h`) can be used to return the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds. The function `srand` (from `<stdlib.h>`) initializes C's random number generator. We often pass the return value of `time` to `srand` so that the program will not generate the same sequence of (pseudo) random numbers. The function `rand` (from `<stdlib.h>`) returns an apparently random number each time it is called.

   More precisely, the following statement can be used to initialize the random number generator:

   ```
   srand((unsigned) time (NULL));
   ```

   Since we are simulating dice throws, we need generate a random number between 1 and 6. This can be done using the following expression

   ```
   rand() % 6 + 1
   ```

   For more details, read the example on page 172-173 of the C textbook.

   Now, write a program that asks the user to specify how many times he/she would like to throw a die. Then, the program simulates throwing the die that many times, and reports how many times you get 1, 2, 3, ... and 6.

   Run your program and use different input value to test it. Try to throw a die 1000 times or more.

   Note that in this program, you need only initialize the random number generator once at the beginning of the program.