Faculty of Computer Science, Dalhousie University *10-Oct-2018*

CSCI 2132 — Software Development

Lab 5: gcc and gdb tools

Location: Teaching Labs Instructor: Vlado Keselj
Time: Thursday

Lab 5: gcc and gdb

Lab Overview

- Learning more about `gcc` and about `gdb`
- Using a program as an example
- Invoking debugger in `emacs`
- More gcc options
- Another C program

In this lab, you will first learn an option of `gcc`. Next you will learn some basic commands of the tool `gdb`, which is a debugger. You will then learn how to invoke the debugger in `emacs`, so that you can edit, compile and debug the program without exiting your editor. During this process, you will learn some hot keys related to multiple windows and buffers in `emacs`. After this, you will learn what options to use if you want `gcc` to use different C language standards. Finally, you will be asked to write a C program to gain more practice in C programming.

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

Step 1: Login and Lab Setup. Login to your bluenose account as before. Go to your checked out copy of your course SVN repository. Based on the previous suggestions, it should probably be in the following directory: `~/csci2132/svn/`*CSID* where *CSID* is your CS user id.

Create and add `lab5` directory to the SVN. You will work in this directory. Commit the change to SVN.

Step 2: gcc -Wall option. We first learn one useful option of `gcc`. Copy the following C source file to your current working directory.

`~prof2132/public/hello.c`

Edit this source file by removing, or commenting out, the line that contains "`return 0;`". Compile it. This will still work and you will not see any warning message regarding this.

We then use the following command to compile:

```
gcc -Wall -o hello hello.c
```

The `-Wall` option enables all the warnings messages about constructions that may be considered questionable under some rules, and that are easy to avoid. Here we will see a warning message related to the missing return statement. Fix your program to avoid the warning by putting back the return statement. Try compiling again and no warning should appear now.

It is often a good idea to use the `-Wall` option, and edit your program to eliminate the warning messages, since sometimes these warning messages are caused by errors in your programs.

SVN submission: Add your file `hello.c` to SVN. Commit your changes to SVN.

Step 3: gcc -g option. Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program. It is an important process in software development. We have discussed debugging and testing strategies in one of the lectures. In this lab, we focus on learning the very basic commands of a debugger called GDB, the GNU project debugger, which is the standard debugger for the GNU software system.

To use `gdb` to debug a C program, we need to use a special option when we compile this program using `gcc`, which is the `-g` option. This option instructs compiler to include symbolic information, such as variable and function names, into the executable program.

Copy the following file into your current working directory:

```
~prof2132/public/numbers.c
```

Compile it using the following command:

```
gcc -g -o numbers numbers.c
```

The executable file `numbers` is now ready to be debugged by `gdb`.

SVN submission: Add the file `numbers.c` to SVN. Commit your changes to SVN.

Step 4: gdb. To run `gdb`, we simply enter `gdb` *program_name*. Try the command:
`gdb numbers`
After you run this command, you will see the command-line interface of `gdb`, which should looke like:
`(gdb)`

You can run the program `numbers` inside gdb. To do this, enter the `gdb` command `run`, or simply `r`. The program will run inside `gdb`, and prompts you to enter a positive integer. Enter 8. After that, `gdb` will finish running `numbers`, it will print '`The result is 21.`', and you will see a line of text saying "program exited normally".

Step 5: Breakpoints. The main approach of debugging with `gdb` is to set breakpoints, which are intentional pausing places in a program. When you set a breakpoint at a certain line in `gdb` and then run this program using `gdb`, the program will pause when this line is executed. You can then use `gdb` commands to print the current values of variables and to check if their values are correct. If there is a bug in the program, you can keep on doing this until you find the line of code that causes the bug. This will help you locate the error.

The syntax of the `gdb` command that sets a breakpoint is "`break` *linenumber*", which sets a break point at a certain line number. You can find out the number of any line by looking at the bottom status bar of the emacs screen, which gives you the line number of the line that has your cursor. You can also list the program from `gdb` using the command `list` or simply `l`.

In this lab, we will not debug an incorrect program. Instead, we will run a correct program using `gdb` to learn how to set breakpoints, to print values of variables and to execute a program line by line. This will help you learn `gdb` commands. You can then start using `gdb` to debug your own code after this.

Assume that you have not quit `gdb`. If you have, repeat the Step 4.

Perform the following steps:

5.a) Print the program listing using command: `l`
5.b) Set a breakpoint at line 8 by entering `break 8` or `b 8`.
5.c) Set a breakpoint at line 20.
5.d) Run the program under `gdb` by entering `run` or `r`.
5.e) Print the value of the variable `value` when the debugger stops at the first breakpoint. You can use the command `print value` or `p value`.

5.f) Execute the next two lines by entering the `gdb` command `step` or `s` twice.
5.g) When the program asks you to input a positive integer, enter `7`.
5.h) Enter a `gdb` command to check whether the value of the variable `k` has been successfully read from stdin.
5.i) Continue the execution of the program until it stops at the next breakpoint by entering `continue`, `cont` or `c`.
5.j) Print the value of the variable `value`.
5.k) Repeat the previous two steps until the program finishes execution.
5.l) Quit the debugger by entering `quit` or `q`.

There are more `gdb` commands than those given above. You can type `help` in the `gdb` command line to see them. They are nicely organized into categories. Try this.

Step 6: Calculated Sequence. Record all the values of variable `value` that you have printed during the above process, in the order that they were printed. Then, enter the sequence of integers at the On-line Encyclopedia of Integer Sequences: `http://oeis.org/`

What is the name of the integer sequence that you find? From this, can you tell what the program computes?

Now read your program again to understand it.

Step 7: gdb in emacs. You can also invoke `gdb` when working in `emacs`. This is good not only because this allows us to edit, compile and debug our code without exiting `emacs`, but also because `emacs` has a `gdb` mode. In this mode, if you divide your screen (vertically, preferably on wide screens) into two windows, having your source code in one window and `gdb` running in the other, there will be an arrow on the `emacs` window for the source code indicating what is the next line to be executed.

To do this, first open `numbers.c` using `emacs`. You can then compile it for debugging in `emacs` (do not forget the `-g` option).

Before using `gdb` in `emacs`, you can refresh your memory of some keys for working with visible windows in `emacs`. You can try:
`C-x 2` to split the window into two windows,
`C-x o` to switch between windows,
`C-x 1` to resume a one-window view,
`C-x 3` to try vertical window split, and
`C-x 1` to again resume a one-window view. Now you can choose one of the split windows, for example:
`C-x 3` to split window vertically.

The following `emacs` commands may also be useful when working with windows:

1. Switching between visible `emacs` windows on the screen: `C-x o`
2. Listing all `emacs` buffers: `C-x b`
3. Killing an `emacs` buffer (window): `C-x k`

Now, when we have the windows split, we can start `gdb` in one of the windows in the following way. Enter `M-x`, and then enter `gdb`. You will see a prompt for the gdb command '`gdb -i=mi a.out`', and you can change it to: `gdb -i=mi numbers`

As before, you can enter commands '`b 8`' and '`b 20`' in gdb to set the breakpoints. You can notice that emacs sets the letter 'B' at the appropriate positions in the source code. You can enter '`r`' to run the program, you will see in the source code that execution stopped at the first break point. Now, use the command '`s`' to step through the program.

After two steps, the program will stop and in the other window we will be prompted to enter a positive integer. With the command '`C-x o`' we can switch to other window and enter 8, for example.

In order to get again the debugger window, we need to do a couple of commands:
`C-x 0` to remove input/output window,
`C-x 3` to split vertically windows again, and
`C-x b *gud-numbers*` to get the gdb window again. Now, you can try the gdb commands that you tried before directly with gdb.

Step 8: gcc option -std. There is one minor detail about 'for' loops in C. In C99, we can declare the loop iterator in the 'for' statement. For example, in the `numbers.c` program, remove the variable '`i`' from the list of variables in the first line inside the main function body. Then, change the first line of the 'for' loop to:

```
for (int i = 1; i < k; i++) {
```

This modification will make the variable '`i` usable in this 'for' loop only.

This is supported in C99, but not in C89 or older versions, which require declarations to precede all the statements in a function body or a block statement.

However, when we compile the modified program using the `gcc` commands that we learned before, we will get an error. Try this to see the error message.

Why? This is because by default, the `gcc` C compiler uses a C language standard called gnu89, which is a GNU dialect of the C language. It contains some of the C99 features only.

How can we compile a C program that uses C99 features that is not in gnu89 then? To do so, use the `-std=c99` option. Enter the following command:

```
gcc -std=c99 -o numbers numbers.c
```

Try this. You can also use other options such as `-g` and `-Wall` at the same time, and the order in which `gcc` options appear does not matter.

Step 9: Compiling binary.c. Copy the file `~prof2132/public/binary.c` to your current directory; i.e., your lab directory.

The `binary.c` file 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 exercise, 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 the next step, we will locate and fix the first bug, while in the step after, we will address the second. If you use emacs, use what you learned already to split your emacs screen in half (remember the control sequences: `C-x 2`, `C-x 3`, `C-x 1`, and `C-x o`). If you wish to invoke the gdb command in the command line, 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.

In the rest of the lab instructions, we assume that you are using emacs, and you compile and debug the program without exiting emacs.

Step 10: Running gdb on 'binary'. Remember that in emacs you can run `gdb` by using `M-x gdb` and then enter the command `gdb binary`.

At the `gdb` prompt (`(gdb)`) enter the command `run` to run the program. 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`. This command can be invoked inside emacs (see gdb Lab).
(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.

Step 11: Finding Another Bug. 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 the previous step, we started debugging from the beginning of this program. This means that we used the linear debugging approach as described in the lectures. 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`, 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 in the previous step). Note: This assumes that you never exited the gdb program that you invoked in emacs. If you start over, then this is not necessary. We also assume that you used the kill command already as suggested in the previous step.
(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 show 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.

Step 12: Testing Program. So far we have fixed these two bugs. It is time to thoroughly test this program. In addition to testing 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.

SVN submission: Add the file `binary.c` to SVN. Commit your changes to SVN.

Step 13: By now, you have finished the required work of this lab. You can work on the next assignment, or practice programming questions from the textbook.

Do not forget to commit all required files to SVN.