1 Introduction

As software projects get bigger and bigger, debugging becomes more and more difficult. You may have gotten used to debugging by printing out every single variable, but trust us, there are much easier ways to fix your programs. With gdb you can pause the execution of your program, print out variables at run time and check for errors in memory usage, all without recompiling.

For a more in depth tutorial on using gdb check out http://www.gnu.org/manual/gdb-5.1.1/html_mono/gdb.html

The first step of this lab is to copy the source code into your directory (cp -r /course/cs123/asgn/gdb ~/course/cs123/).

2 Background and command summary

Once you learn a few of the basic commands of gdb, you will be able to do a lot. One important command to know is help. If you type help you will get a list of broad help topics. If you wanted to get more information about running programs within gdb typing help would show you that running is a help topic. You can then type help running to get even more sub-topics related to running programs. It will list all of the other gdb commands that you are likely to use. help <command> will usually give you a great deal of information about that command and its use. The gdb help system is a great resource, do not forget about it!

2.1 Starting gdb and running programs

Loading up a file in gdb is relatively simple. Simply run gdb <executable> to start gdb up on the executable. In order to use gdb to its fullest you MUST have compiled your program with the -g compiler option. When you are in gdb and have a program loaded, these commands will help you get started.

- help running
  Will give a list of commands related to running the program.
- run <args>
  Will run the current program being debugged with the given arguments.
- set args <args>
  Will permanently set the arguments being sent to the program.

So, for instance, to answer the problems at the end of the lab you might run gdb debugme to start gdb with the debugme executable being debugged. After this, you can type run to run the debugme executable within gdb.

One thing to note is that gdb has a very nice tab completion system. It will tab complete commands, file names, and even symbols within your program! One slight caveat to this is that to complete C++ symbols you need to start an expression with a single quote when you tab complete. So, for instance, when doing the exercise at the end of the lab if you wanted to break inside of a method of the Counter class you could type:

break 'Counter

and then hit the tab key twice for gdb to give you a list of possible methods. Then if you continue typing:

break 'Counter::c'
you can hit the tab key again and **gdb** will tab complete the function to be the following:

```
break 'Counter::count(void)'
```

Another tip is that when invoking a **gdb** command you only have to type as many letters as is necessary to disambiguate that command from any others. For instance, if you type 'whe' and hit enter **gdb** will assume you are just being lazy and you want to run the 'where' command.

## 2.1.1 Commands to control execution

Here are some basic commands that will let you pause the execution of your program and examine data.

- **break** `<location>`
  This command sets a breakpoint at a specified location. Some examples:
  - `break main` – Sets a breakpoint in the main() function.
  - `break Foo::myFunc` – Sets a breakpoint in the Foo::myFunc method.
  - `break main.C:100` – Sets a breakpoint in the main.C file, at line number 100

- **break** `<location>` if `<condition>`
  This command sets a breakpoint at a specified location with a conditional requirement such that the breakpoint is only hit if the conditional is true. For example:
  - `break myFunc if m_variable==0`

- **condition** `<breakpoint number>` `<condition>`
  This command sets an existing breakpoint to be conditional so that execution of the program only halts at the specified breakpoint when the the condition specified is true.

- **info** `<action type>`
  Displays all breakpoints or auto-display expressions in effect. The 'action type' argument can either be 'breakpoints' or 'display' depending upon whether you wish to examine breakpoints or auto-display expressions. You can even use 'locals' to show all local variables of the current stack frame. Try **help info** for more information.

- **step**
  If stopped in execution, this will execute the next line of code. If this line is a function call, it will enter the function call. It is common to repeatedly call step to advance slowly through your program’s execution.

- **next**
  If stopped in execution, this will execute the next line of code, not stepping into any function calls. It is common to repeatedly call next to advance slowly through your program’s execution.

- **continue**
  If stopped in execution, will begin to execute normally and run until the next breakpoint or until the program exits. It is common to call **cont** when you have finished using step or next to examine things closely and want to have the program execute at full speed.

- **delete** `[<action type>]` `<number>`
  This deletes one of the breakpoints or auto-display expressions displayed by the info command. The 'action type' argument can either be 'breakpoints' or 'display' depending on whether you are deleting a breakpoint or auto-display expression. If the 'action type' argument is not given that it is assumed you are deleting a breakpoint.
• clear [<location>]
  This clears a specified location of all breakpoints. If no line number is specified this clears all break-
  points.

• call <function>(args)
  Instructs gdb to call the given function, with the given arguments. Use print to show the return value
  (see next section).

2.2 Displaying and Naming Data

Once you reach a certain point in the execution of a program, it is often useful to be able to print out
information about the execution of the program up to that point. For example, say you have some functions:
funcOne calls funcTwo which calls funcThree. If you stop at a breakpoint in funcThree, you can print
out the stack trace using the where command. This will print out a list of the calls which led to reaching
funcThree, and the values of their arguments.

While you are stopped in funcThree, you can also print out the values of all variables within the scope of
the function, which may include local variables, class variables, and memory addresses, by using the print,
display, and examine commands. Within the scope of funcThree, you can’t access the variables in the
scope of funcTwo. However, you can move between stack frames by using the up and down commands. Using
the up command will move you from the scope of funcThree to the scope of funcTwo, where you can view
the stack trace and variables of funcTwo. Using the down command will bring you back to the scope of
funcThree.

The important commands to know for looking at data are:

• where
  Prints a list of all stack frames, along with the function and argument values for each stack frame.

• up [<number>]
  Move up one stack frame. An optional argument can move up multiple frames.

• down [<number>]
  Move down one stack frame. An optional argument can move down multiple frames.

• frame <number>
  Jump to an exact frame number.

• print <expression>
  Prints the value of the given expression. The expression can be a function call, local variable, or a
  constant.

• list
  Prints the code that surrounds the current point of execution.

• display <expression>
  Display the value of an expression every time execution is halted at a breakpoint.

It should be noted that all the above expressions may be arbitrarily complex and contain any math, bit
operations, etc. They need not use variables from the program (e.g. print (1291 & 15) << 2).
2.3 Leak Checking and Valgrind

Memory leaks pose a serious problem and are notoriously difficult to find, as they often don’t cause consistent problems with your program. To this end, you are encouraged to use a tool called valgrind.
To check for memory leaks, type

```bash
valgrind --leak-check=yes <program name> <program arguments>
```

at a prompt. This will check for misuse of memory and memory leaks as you run the program. Note that misuse of memory is very liberally defined according to valgrind! You will likely (or at least hopefully) find that most of the errors are XLib’s fault. Don’t panic if you get a lot of memory errors – the point of using this tool is to figure out your memory leaks. Once your program terminates, valgrind will print out a list of memory leaks, and their locations in your program. Use this to help you fix the leaks in your code. Be sure to compile your program with debugging flags turned on.

2.4 Abbreviations

- n is equivalent to next
- s is equivalent to step
- c is equivalent to cont
- wh is equivalent to where
- do is equivalent to down
- b is equivalent to break
- p is equivalent to print
- l is equivalent to list

3 Extended Example

The following is an example of the first few commands issued after a program crashes.

- > gdb buggy
  Begin debugging the program that has the bug.
- (gdb) run
  Run the program within gdb. Interact with the program to try to recreate the conditions that made the program crash earlier. After run is called, in all likelihood, the program will crash at the same point that it did before. At this point gdb will halt execution of the program and return you to the gdb prompt.
- (gdb) list
  This lists the 10 lines of code around the current line that we are stopped at.
- (gdb) where
  The where command will show us the complete call stack.
- (gdb) down 2
  This jumps down two function calls.
- (gdb) print tempInt
  This prints out the value of tempInt which might be a local variable in the current function being examined.
4 Exercises

Compile the given code in /course/cs123/src/gdb. You are to answer all the following questions WITH-OUT modifying the program in any way (you may look at the code, however). All work should be done in the debugger.

1. What is the call stack when QuietAlarm::goOff() gets called?

2. Set a breakpoint in Counter::count(). Set m_count to be displayed every time you stop at a breakpoint. Continue running the program and stopping in count() until m_count is 5. Then remove the display of the variable, remove the breakpoint and continue execution.

3. When the program is about to terminate, what is the value of QuietAlarm's m_goOffCount?

4. When m_count is 75, what is m_relayCount?

5. When m_count is 50, how many alarms are on the m_alarms list?

6. What is the value of QuietAlarm's m_goOffCount variable when the NoisyAlarm goes off at m_count 32?

7. Run this program while checking for memory leaks. Are there any memory leaks? Where did they come from?