**CS 382E: Lab 4**
**Date: September 27, 2022**

Philippos Mordohai and Shudong Hao
Department of Computer Science
Stevens Institute of Technology

## 1 Objective

In this lab, we are going to get familiar with GNU debugger (`gdb`), which allows us to set breakpoints and then inspect the internal states of a program. We will use `gdb` to debug C programs, but it can also be used on assembly programs, as we will see in a future lab.

## 2 Installation

Generally Unix distributions come with `gdb`, but it can be installed using the following commands in the terminal:

```
sudo apt-get update
sudo apt-get install gdb
```

For more details refer to http://www.gdbtutorial.com/tutorial/how-install-gdb.

On macOS with the M1 chipset, you can use `lldb` instead of `gdb`, because the latter does not support the M1 chipset yet. On macOS on an Intel processor, consider the instructions on https://gist.github.com/danisfermi. Please use the Virtual Machines in CS 382.

## 3 Debugging

In general, we use the following command to compile C code:

```
gcc prog.cpp -o myexec
```

(Try to avoid the default `a.out` for your executable programs. Give them a meaningful name.)

To enable debugging, the `-g` option should be included.

```
gcc -g prog.cpp -o myexec
```

We can start the debugger on our program in one of the following two ways. We can type the following command

```
gdb myexec
```

We can also enter the debugger first by using

```
gdb
```

and then typing the name of the executable `myexec`.

To become familiar with a few commands in `gdb`, try them on your code, but note that some will not work yet because they rely on other `gdb` commands being run first.

```
help command        # ask gdb to tell you about the named gdb command
run [arg1 ... argn] # run your code (and add command line arguments if required)
```

After issuing the `run` command, the program will be executed from the beginning to the end. This looks useless because it is the same as running the program from the terminal directly. The advantage of a debugger is that we can set **breakpoints** in the code, causing the program to pause when it reaches these points, enabling us to see what is going on.

### 3.1 `break`

We need the debugger to pause at a certain point in the code so that we can investigate the program. To set a breakpoint, we will use the `break` command.

```
break prog.c:12
```

In this example, we are setting a breakpoint in file `prog.c` at line number 12. If we are interested in a particular function we can set a break point at that function and the debugger will pause every time the function is called:

```
break function_name
```

After setting breakpoints, when we give the `run` command, the program will be paused at the breakpoints.

### 3.2 `continue`

To move on to next break point we can use the command `continue`, or simply `c`.

### 3.3 `step` and `next`

To proceed by a single step, we can either use `step` or `next`, but there is a subtle difference between them. If the next line of code is a function call, `next` will consider it as a single instruction and will execute the function all at once. On the other hand, `step` will go through the lines of the function, providing more fine-grained control than `next`.

If we suspect that the function has something to do with the error or bug, we might want to `step` into the function. But if we are sure that the function is correct, we can hit `next` which will run the function and bring us to the statement after the function call.

### 3.4 `print` and `display`

To print a value of a variable we can use `print` command.

```
print var   # var is a variable;
print *ptr  # ptr is a pointer.
```

The `print` command will print the value only once. If you want to print the value each time your are in the scope where the variable is defined you can use the `display` command.

### 3.5 `watch`

Whereas breakpoints interrupt the program at a particular line or function, *watch points* act on variables. They pause the program whenever a watched variable's value is modified.

```
watch var
```

Note that each time we make any change in the code, we need to stop `gdb` and recompile the program to generate the updated executable. Then run `gdb` with this new executable file.

## 4 Lab Task

Download `midpoint_gold.c` provided on Canvas. Use `gdb` to perform the following checks on the program:

(1) Confirm that the number of arguments to the program is 3.

(2) Confirm that `argv[1]-argv[3]` are pure numbers. Note that `atof()` will extract the first floating point number possible from the `argv[k]` string and stop when a letter etc. is encountered.

(3) Demonstrate that "2.3xyz" can be provided as the upper integration bound and the program will still work.

(4) Provide invalid bounds and check that the program exits appropriately using the `gdb step` command.

(5) Set a breakpoint in `func()` to inspect the value of `f` once.

(6) Set the number of rectangles to 5 and print the values of result within one execution of `midpoint`.

(7) Set one or more breakpoints to see all values of the approximation. (Set the number of rectangles so that the program takes 4 iterations to converge.)

Items 1-4 are worth 10 points each, while items 5-7 are worth 20 points each.

## 5 Deliverable

A pdf file with screenshots, **not photographs,** demonstrating the steps specified above and **brief descriptions** of what each screenshot shows.