CS 261 Recitation 3: GDB Practice & Complexity Analysis

In order to get credit for the recitation, you need to be checked off by the end of recitation. For non-zero recitations, you can earn a maximum of 3 points for recitation work completed outside of recitation time, but you must finish this recitation before the next recitation. For extenuating circumstance, contact your Instructor.

Group work, and individual work are highlighted

Recitation 3 Grade Breakdown:

• Part 1: GDB practice	6 pts
• Part 2: Detect Anagram and Complexity Analysis	4 pts

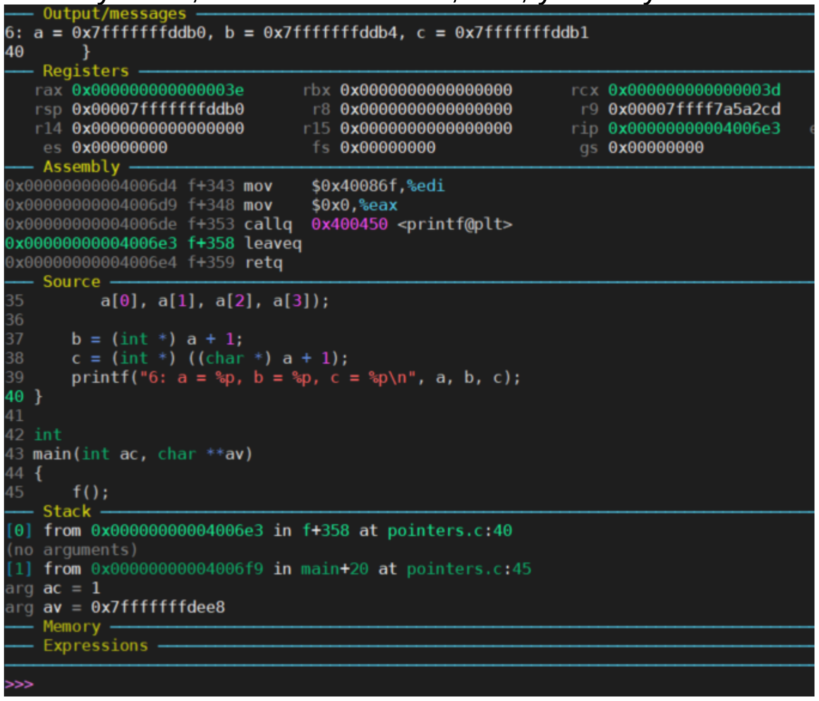
Download and unzip the start code for this recitation:

wget https://classes.engr.oregonstate.edu/eecs/summer2022/cs261-001/recitations/rec3.zip

Part 1: GDB practice

Step 0 (Optional): GDB Setup

If you prefer a more informational GDB interface (see below) with register values, source code, assembly code, stack information, etc., you may run the following script:

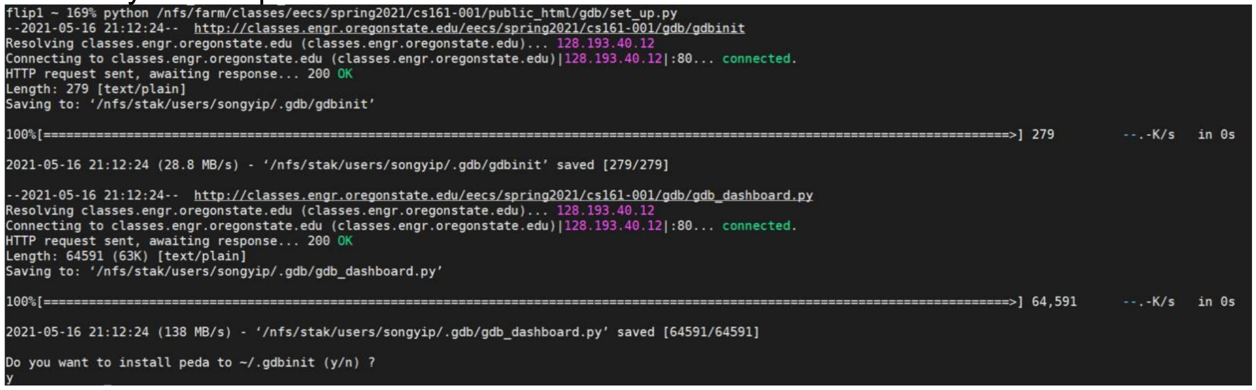


```
─── Output/messages ─────────────────────────────────────────────
6: a = 0x7fffffffddb0, b = 0x7fffffffddb4, c = 0x7fffffffddb1
40      }
─── Registers ───────────────────────────────────────────────────
   rax 0x000000000000003e      rbx 0x0000000000000000      rcx 0x000000000000003d
   rsp 0x00007fffffffddb0       r8 0x0000000000000000       r9 0x00007ffff7a5a2cd
   r14 0x0000000000000000      r15 0x0000000000000000      rip 0x00000000004006e3
    es 0x00000000               fs 0x00000000               gs 0x00000000
─── Assembly ────────────────────────────────────────────────────
0x00000000004006d4 f+343 mov    $0x40086f,%edi
0x00000000004006d9 f+348 mov    $0x0,%eax
0x00000000004006de f+353 callq  0x400450 <printf@plt>
0x00000000004006e3 f+358 leaveq
0x00000000004006e4 f+359 retq
─── Source ──────────────────────────────────────────────────────
35          a[0], a[1], a[2], a[3]);
36
37      b = (int *) a + 1;
38      c = (int *) ((char *) a + 1);
39      printf("6: a = %p, b = %p, c = %p\n", a, b, c);
40  }
41
42  int
43  main(int ac, char **av)
44  {
45      f();
─── Stack ───────────────────────────────────────────────────────
[0] from 0x00000000004006e3 in f+358 at pointers.c:40
(no arguments)
[1] from 0x00000000004006f9 in main+20 at pointers.c:45
arg ac = 1
arg av = 0x7fffffffdee8
─── Memory ──────────────────────────────────────────────────────
─── Expressions ─────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────
>>>
```

In your home directory, type:

```
python /nfs/farm/classes/eecs/spring2021/cs161-001/public_html/gdb/set_up.py
```

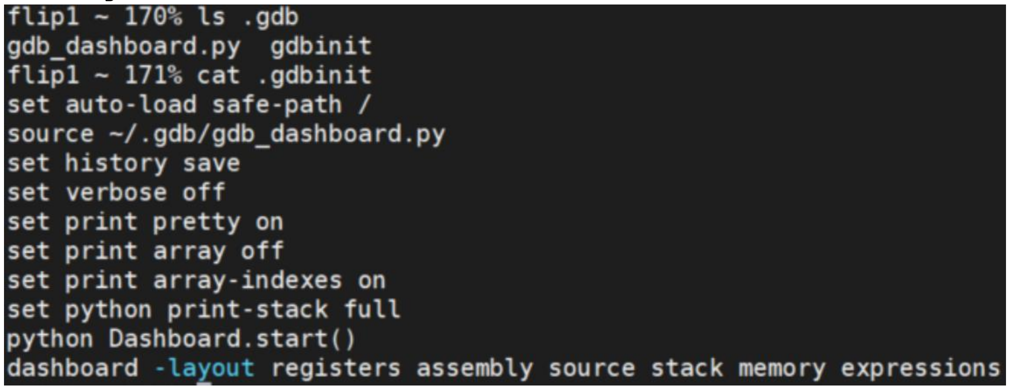
Answer ‘y’ to the question:



```
flip1 ~ 169% python /nfs/farm/classes/eecs/spring2021/cs161-001/public_html/gdb/set_up.py
--2021-05-16 21:12:24--  http://classes.engr.oregonstate.edu/eecs/spring2021/cs161-001/gdb/gdbinit
Resolving classes.engr.oregonstate.edu (classes.engr.oregonstate.edu)... 128.193.40.12
Connecting to classes.engr.oregonstate.edu (classes.engr.oregonstate.edu)|128.193.40.12|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 279 [text/plain]
Saving to: ‘/nfs/stak/users/songyip/.gdb/gdbinit’

100%[=====================================================================================================================>] 279         --.-K/s   in 0s

2021-05-16 21:12:24 (28.8 MB/s) - ‘/nfs/stak/users/songyip/.gdb/gdbinit’ saved [279/279]

--2021-05-16 21:12:24--  http://classes.engr.oregonstate.edu/eecs/spring2021/cs161-001/gdb/gdb_dashboard.py
Resolving classes.engr.oregonstate.edu (classes.engr.oregonstate.edu)... 128.193.40.12
Connecting to classes.engr.oregonstate.edu (classes.engr.oregonstate.edu)|128.193.40.12|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 64591 (63K) [text/plain]
Saving to: ‘/nfs/stak/users/songyip/.gdb/gdb_dashboard.py’

100%[=====================================================================================================================>] 64,591      --.-K/s   in 0s

2021-05-16 21:12:24 (138 MB/s) - ‘/nfs/stak/users/songyip/.gdb/gdb_dashboard.py’ saved [64591/64591]

Do you want to install peda to ~/.gdbinit (y/n) ?
y
```

Once setup successfully, you will have a `.gdb` folder and a `.gdbinit` file under your home directory, and you can verify it with:

```
ls .gdb
cat .gdbinit
```



```
flip1 ~ 170% ls .gdb
gdb_dashboard.py  gdbinit
flip1 ~ 171% cat .gdbinit
set auto-load safe-path /
source ~/.gdb/gdb_dashboard.py
set history save
set verbose off
set print pretty on
set print array off
set print array-indexes on
set python print-stack full
python Dashboard.start()
dashboard -layout registers assembly source stack memory expressions
```

Step 1: Demo

Before you start working on using GDB yourself, your TAs will demonstrate how to use GDB to debug a buggy program, `buggy-pointers.c`. Watch carefully as they explain how to run the program with GDB and how to use GDB’s various commands, and follow along yourself in the source code as they debug.

To use GDB, you need to compile your program with a debugging (`-g`) flag

```
gcc --std=c99 -g [some_program.c] -o [exe_name]
```

Then, run gdb with the executable:

```
gdb ./[exe_name]
```

Some important commands to watch for during the demo (links to command documentation are included):

- `run` (`r`) – starts your program from the beginning. Command line arguments to your program can be specified with the `run` command.
  `run --args [args]`

- `break` (`b`) – tells GDB to pause the execution of your program once it reaches a specified line in your source code. This is called setting a ***breakpoint***.
  ```
  break [file_name]: [line_num]
  break [function_name]
  ```

- `list` (`l`) – prints out the lines of source code near the one currently being executed or near a specified location.

- `print` (`p`) – prints out the GDB value stored in a specified variable, etc.
  ```
  print [var_name or function_name]
  ```
  You may also print out the address of a specified variable.
  ```
  print &[var_name]
  ```

- `step` (`s`) – tells GDB to execute the very next line of code when it's paused at a breakpoint. If the next line of code is inside a function call, the `step` command enters that function.

- `next` (`n`) – a lot like the `step` command; tells GDB to execute the very next line of code when it's paused at a breakpoint. However, if the next line of code is inside a function call, the `next` command runs that function without entering into it.

  As you may notice, each statement may contain multiple assembly instructions. You may also run those assembly instructions one by one by "next instruction" or "`ni`"

- `watch` – tells GDB to pause whenever the value of a specified variable changes and to print out the change in that variable's value. This is called setting a ***watchpoint***.
  ```
  watch [var_name]
  ```

- `continue` (`c`) – tells GDB to resume normal execution of the program from the line of code where it's currently stopped until the next breakpoint, or the end of the program.

- `backtrace` (`bt`) – prints a backtrace, which is the sequence of function calls (called ***frames***) that brought the program to the current line of code being executed

- x/100wx [address or register] – read memory
  x – Examine
  100 – 100 values
  w – sized as word (w, 4 bytes) / b – 1 byte / g – 8 bytes
  x – In hexadecimal (x) / d – decimal

(3 pts) Step 2: Use GDB to resolve the segmentation fault

Try to use GDB to debug the program `buggy-list-sort.c`. The program in `buggy-list-sort.c` specifically generates a short linked list containing random integers and tries to sort it using bubble sort. However, something in the code is broken, and the program crashes with a segmentation fault before the sorting is complete. If you were able to successfully debug the program, make sure to describe all of the bugs you found and how you fixed them in `list_sort.txt`.

(3 pts) Step 3: Use GDB to resolve the logic error

Try to use GDB to debug the program `buggy-array-sort.c`. This program generates a small array of random integers and tries to sort it using insertion sort. However, something in the program is broken, and the array is not correctly sorted. If you were able to successfully debug the program, make sure to describe all of the bugs you found and how you fixed them in `array_sort.txt`.

Part 2: Detect Anagram and Complexity Analysis

(2 pts) Step 1: Implement `check_anagram()`

One string is an anagram of another if the second is simply a rearrangement of the first. For example, "**heart"** and "**earth"** and are anagrams. The strings "**python"** and "**typhon"** are anagrams as well. Assuming that the two strings in this question are of equal length and that they are made up of symbols from the set of 26 lowercase letters (a-z). In `anagram.c`, Write a function `check_anagram()` that will take two strings and return 1 if they are anagrams, and return 0 if they are not.

(2 pts) Step 2: Analyze `check_anagram()`

Analyze the run-time complexity of the `check_anagram()` function you wrote using Big O notation, and put your explanation and result in a text file named `anagram_analysis.txt`.

Extra credit: (2 pts) Step 3: Optimize `check_anagram()`

There are multiple ways to tell whether the two given strings are anagrams or not. For extra credit, implement the function `check_anagram()` in `anagram.c` to detect anagrams with O(n) complexity.

Make sure you get checked off by showing them the output of your program and your group work before the end of your recitation section.

For backup purposes, please submit your work for this recitation (including all documents/text files for group work, and programs) to TEACH.