CS 162 LAB #6 – GDB & Practice Classes

In order to get credit for the lab, you need to be checked off by the end of lab. You can earn a maximum of 3 points for lab work completed outside of lab time, but you must finish the lab before the next lab and get checked off with your Instructor or TAs during their office hours. For extenuating circumstances, contact your TAs and the instructor.

This lab is worth 15 points total. Here's the breakdown:

- Part 1: Worksheet (5 pts)
- Part 2: GDB Practice (Group Work: 2pts)
- Part 3: Classes
  - Implement a class with constructors, accessors, and mutators (Individual Work: 5pts)
  - Design the class composition (Individual Work: 2pts)
  - create makefile (Individual Work: 1pts)

(5 pts) Part 1: Worksheet

This session will be led by your lab TAs. Please follow their instructions, participate, and complete worksheet 5:

https://classes.engr.oregonstate.edu/eecs/winter2024/cs162-001/labs/WS6.docx (pdf version)

(2 pts) Part 2: GDB Practice

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" the program while it executes -- or what the program was doing at the moment it crashed.

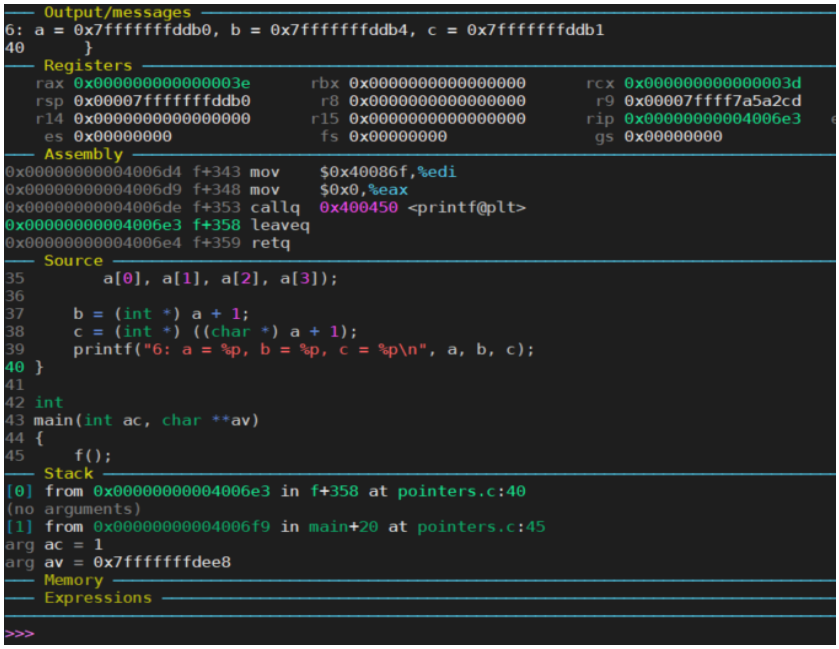
GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB Manpage is a good source of information, i.e. `man gdb`

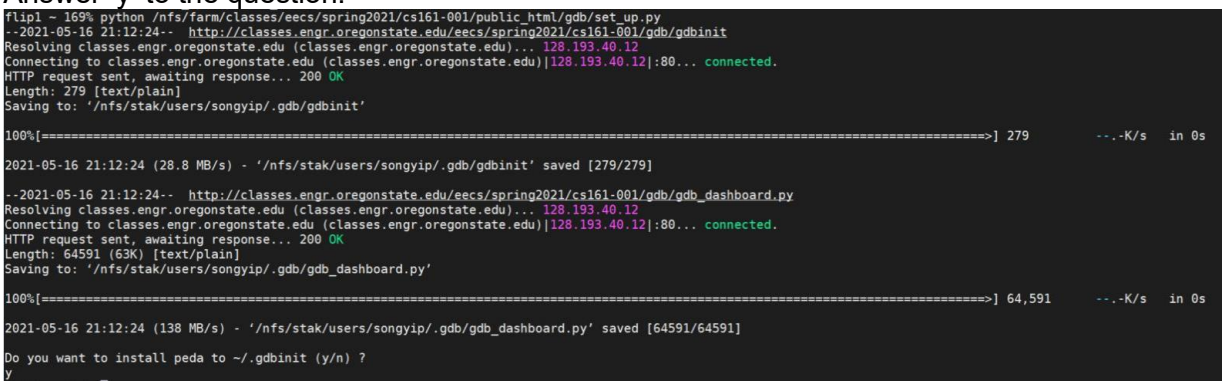
Step 0 (Optional): GDB Setup

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:



In your home directory, type:
`python /nfs/farm/classes/eecs/spring2021/cs161-001/public_html/gdb/set_up.py`

Answer 'y' to the question:



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: Using GDB (TA Demo). **Make sure you are able to follow every step in the demo**
To start debugging your program, you need to compile it with debugging symbols, this is accomplished with the **`-g`** flag:

```
g++ filename.cpp -g -o filename
```

Let's start with a simple program that gets a line of text from the user, and prints it out backwards to the screen: (You may get the file at https://classes.engr.oregonstate.edu/eecs/winter2024/cs162-001/labs/debug.cpp using wget command)

```
#include <iostream>
#include <cstring>

using namespace std;

int main(){
   char input[50];
   int i = 0;

   cin >> input;

   for(i = strlen(input); i >= 0; i--){
      cout << input[i];
   }
   cout << endl;

   return 0;
}
```

compile and start the debugger with: (inputs are highlighted)

```
g++ debug.cpp -g -o debug
gdb ./debug   (start another session which will run gdb)
```

GDB Execution

```
flip1 ~/cs161-010/private 169% gdb ./debug
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /nfs/farm/classes/eecs/winter2020/cs161-010/private/debug..
.done.
(gdb)
```

debug.cpp

```
1  #include <iostream>
2  #include <string.h>
3
4  using namespace std;
5
6  int main(){
7
8      char input[50];
9      int i = 0;
10
11     cin >> input;
12
13     for(i = strlen(input); i >= 0; i--){
14         cout << input[i];
15     }
16     cout << endl;
17
18     return 0;
19 }
```

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

1. **The `break` Command:**
   `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]
   ```

   Continuing with our example lets set up a break point at line 9, just before we declare `int i = 0;`

   ```
   >>> break 9
   ```

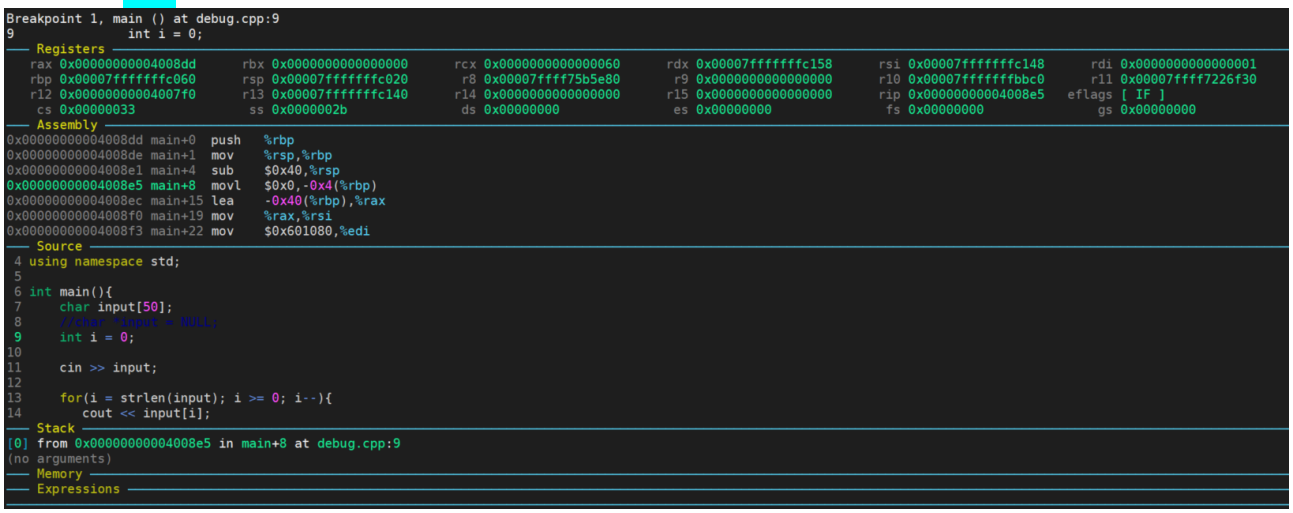
```
Breakpoint 1 at 0x4008ed: file debug.cpp, line 9.
>>>
```

2. **The `run` Command:**
   `run` (`r`) – starts your program from the beginning. This will run your program as you normally would outside of the debugger, until it reaches a break point line. *Command line arguments to your program can be specified with the `run` command.

   ```
   run --args [args]
   ```

   At this moment, you will have been returned to the `gdb` command prompt. (Using `run` again after your program has been started, will ask to terminate the current execution and start over)

   From our example:
   `>>> run`



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

4. **The `print` Command:**
   `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]`

   Let's look what the value of i is now:

   ```
   >> print i
   $1 = 0
   ```

5. **The `next(n)` and `step(s)` Commands:**
   `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"

Now in our example, we will "next" to the beginning of the next instruction.

```
>> next
11      cin >> input;
```

What happened here? We weren't returned to the gdb prompt. Well, the program is inside cin, waiting for us to input something.

Input string here, and press enter.

6. **The `continue` Command**

`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.

Let's continue to the end of the program now:

```
>>> continue
Continuing.
olleh

[Inferior 1 (process 9059) exited normally]
>>>
```

Here we've reached the end of our program, you can see that it printed in reverse "input", which is what was fed to cin.

7. **The `display` and `watch` Commands:**

`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]
```

`display` will show a variable's contents at each step of the way in your program. Let's start over in our example. Delete the breakpoint at line 9

```
>>> del break 1
```

This deletes our first breakpoint at line 9. You can also clear all breakpoints w/ clear.

Now, let's set a new breakpoint at line 14, the cout statement inside the for loop

```
>>> break 14
Breakpoint 2 at 0x40094c: file debug.cpp, line 14.
```

Run the program again with the **`run`** command, and enter the input. When it returns to the gdb command prompt, we will display input[i] and watch it through the for loop with each next or breakpoint.

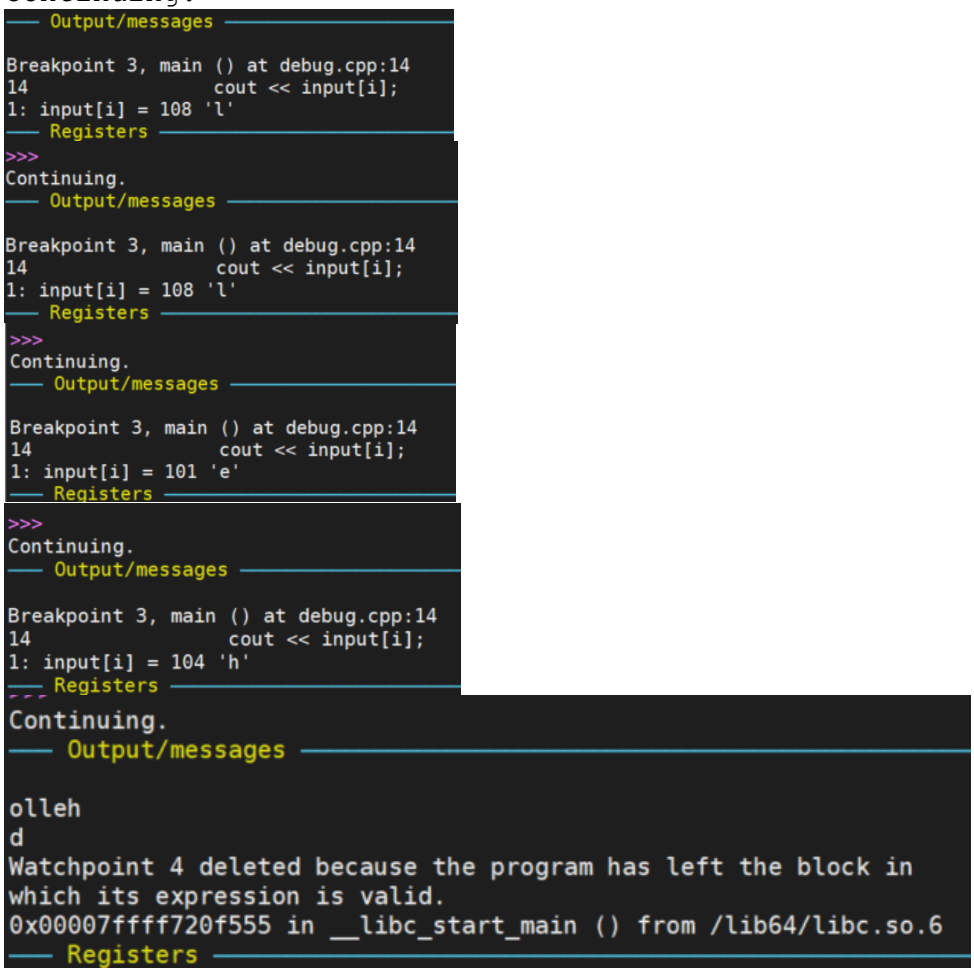
```
Breakpoint 2, main () at debug.cpp:14
14              cout << input[i];
>>> display input[i]
1: input[i] = 0 '\000'
>>> next
13      for(i=strlen(input);i>=0;i--) {
1: input[i] = 0 '\000'
```

```
>>> next

Breakpoint 2, main () at debug.cpp:14
14              cout << input[i];
1: input[i] = 111 'o'
>>> next
14      for(i=strlen(input);i>=0;i--) {
1: input[i] = 111 'o'

 >>> next
```

Here we stepped through the loop, always looking at what `input[i]` was equal to.

We can also watch a variable, which allows us to see the contents at any point when the memory changes.

```
>>> watch input
Watchpoint 2: input
>>> continue
Continuing.
```



```
>>>
Continuing.
── Output/messages ──────────────────────────────────────
[Inferior 1 (process 13902) exited normally]
>>> q
flip1 ~ 183%
```

Type `q` and hit enter to exit from GDB.

8. **The `backtrace (or bt)` Command**
   `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

   Let's modify our program just a little so that it will crash:

```
#include <iostream>
#include <cstring>

using namespace std;

int main(){
     char *input = NULL;
     int i = 0;

     cin >> input;

     for(i = strlen(input); i >= 0; i--){
        cout << input[i];
     }
     cout << endl;

     return 0;
}
```

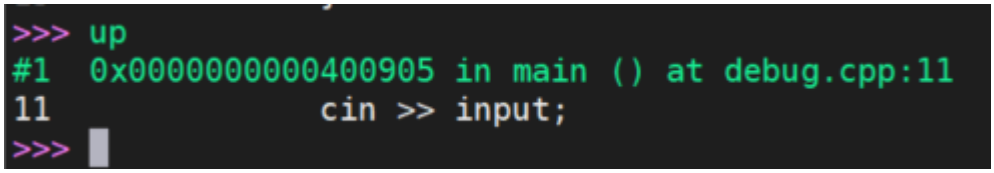
Here we've changed input to be a pointer to a char and set it to NULL to make sure it doesn't point anywhere until we set it. Recompile and run `gdb` on it again to see what happens when it crashes.

```
>>> r
Starting program: …
hello
```

```
Program received signal SIGSEGV, Segmentation fault.
std::operator>><char, std::char_traits<char> > (__in=warning: RTTI symbol not found for class 'std::istream'
..., __s=0x0) at ../../../../../libstdc++-v3/src/c++98/istream.cc:247
247                       *__s++ = __traits_type::to_char_type(__c);
```

```
>>> bt
#0  std::operator>><char, std::char_traits<char> > (__in=warning: RTTI symbol not found for class 'std::istream'
..., __s=0x0) at ../../../../../libstdc++-v3/src/c++98/istream.cc:247
#1  0x0000000000400905 in main () at debug.cpp:11
```

We see at the bottom, two frames. #1 is the top most frame and shows where we crashed. Use the up command to move up the stack.



Here we see line #11

```
11 cin >> input;
```

The line where we crashed.

9. Examine the memory (very useful)
   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

Here are some more tutorials for gdb:

- http://www.cs.cmu.edu/~gilpin/tutorial/
- https://sourceware.org/gdb/current/onlinedocs/gdb/

Part 3: Practice Classes

In Assignment 3, you need to create `Flight`, `Airport`, and `Manager` classes. Seems a lot, right? But don't worry, this lab serves to get you a head start on it!

First, download the skeleton code for assignment 3:

Command to download:

```
wget https://classes.engr.oregonstate.edu/eecs/winter2024/cs162-001/assignments/assign3.zip
```

Command to extract:

```
unzip assign3.zip
```

(5 pt) Step 1: Implement constructors, accessors, and mutators for one class

Start working on the .h and .cpp files for one of the classes with the appropriate members (all being private), mutator functions, accessor functions, and constructors. Add **const** keyword when appropriate!

*Note: In real life, we create mutators and accessors only if we need them, but to give you more practices, let's create a mutator and accessor for each member variable of the class in this assignment.

For example, **here are some prototypes for the default constructor, mutators, accessors, and some other useful functions for the Flight class to get you started.**

```
Flight();              //Flight constructor

void set_flight_num (string);

void set_curr_loc (string);

void set_dest(string);
```

```
void set_num_pilots(const int);
void set_pilots(string*);

string get_flight_num () const;
string get_curr_loc () const;
string get_dest() const;
int get_num_pilots() const;
string* get_pilots() const;

void print_flight() const; //print the flight object
```

(2 pts) Step 2: Class Composition

Now, let's figure out how classes interact with each other**.** On a sheet of paper, write down the relationship between classes involved in this assignment (i.e. Airport "has-a" Flight). Besides, explain how you are going to implement the "has-a" relationship.

(1 pts) Step 3: Create `makefile`

Create a Makefile that compiles all of your .cpp files and makes an executable.

Remember, you will not receive lab credit if you do not get checked off before leaving each lab. Once you have a zero on a lab, then it cannot be changed because we have no way of knowing if you were there or not.

Show your completed work and answers to the Instructor or the TAs for credit. You will not get points if you do not get checked off!

Submit your work to TEACH for our records **(Note: you will not get points if you don't get checked off with your instructor or a TA!!!)**

1. Create a **zip file** that contains all files you've created in this lab:
2. Transfer the tar file from the ENGR server to your local laptop.
3. Go to TEACH.
4. In the menu on the right side, go to **Class Tools → Submit Assignment**.
5. Select **CS162 Lab6** from the list of assignments and click "**SUBMIT NOW**"
6. Select your files and click the Submit button.