Lab 1 **C and GDB**

Objective: **Practice with pointers, strings, and structs**
Debugging skills: compiler warnings, assert statements, and GDB

Setup

You already have your local repository of this course. You just need to explore this lab using cd and ls commands. Go to fa21-lab-starter/lab01 and list all the exercises.

Exercise 1

Learning Goals: Practice the C programming concepts you have learned in lecture: strings, structs, and pointers. Introduction to assert()

Part 1

1. Implement the function num_occurences in exercise1/ex1.c. The directions for completing the function can be found in the comments above the function. test_ex1.c will be used to test your code.

   Compiling and Running a C Program
   In this lab, we will be using the command line program gcc to compile programs in C. Use the following command to compile the code for Part 1

   ```
   gcc ex1.c test_ex1.c
   ```

   This compiles ex1.c and test_ex1.c into an executable file named a.out. If you've taken CS61B or have experience with Java, you can kinda think of gcc as the C equivalent of javac. This file can be run with the following command:

   ```
   ./a.out
   ```

   The executable file is a.out, so what is the ./ for? Answer: when you want to execute an executable, you need to prepend the path to the name of the executable. The dot refers to the "current directory." Double dots (..) would refer to the directory one level up.
   gcc has various command line options which you are encouraged to explore. In this lab, however, we will only be using -o, which is used to specify the name of the executable file that gcc creates. By default, the name of the executable generated by gcc is a.out. You can use the following commands to compile ex1.c

into a program named ex1, and then run it. This is helpful if you don't want all of your executable files to be named a.out.

```
gcc -o ex1 ex1.c test_ex1.c./ex1
```

Assert

The Linux man pages serve as a manual for various standard library and operating system features. You can access the man pages through your terminal.

Type the following line into your terminal to learn about assert

```
man assert
```

To exit the man pages, press 'q'.

2. Think of a scenario that is not tested by the current test cases. Create one additional test case to test this scenario.

Part 2

1. Implement the function compute_nucleotide_occurrences in ex1.c. Look at ex1.h to see relevant struct definitions.
   Hint: You may be able to reuse num_occurences
2. Think of a scenario that is not tested by the current test cases. Create one additional test case to test this scenario.

Exercise 2

Learning Goals: Get familiar with basic GDB commands

Part 1: Compiler Warnings

1. Read over the code in exercise2/pwd_checker.c.
2. Learn about compiler warnings and the importance of resolving them. This section will resolve bug(s) along the way. Make sure to fix the bug(s) before moving on to the next section.
    - Compiler warnings are generated to help you find potential bugs in your code. Make sure that you fix all of your compiler warnings before you attempt to run your code. This will save you a lot of time debugging in the future because fixing the compiler warnings is much faster than trying to find the bug on your own.
        - Compile your code
          ```
          gcc pwd_checker.c test_pwd_checker.c -o pwd_checker
          ```
          The -o flag names your executable pwd_checker.
    - You should see 4 warnings. When you read the warning, you can see that the first warning is in the function check_upper. The warning states that we are doing a comparison between a pointer and a zero character constant and prints out the line where the warning occurs.
    - The next line gives us a suggestion for how to fix our warning. The compiler will not always give us a suggestion for how to fix warnings. When it does give us a suggestion, it might not be accurate, but it is a good place to start.
    - Take a look at the code where the warning is occuring. It is trying to compare a const char * to a char. The compiler has pointed this out as a potential error because we should not be performing a comparison between a pointer and a character type.
        - This line of code is trying to check if the current character in the password is the null terminator. The code is currently comparing the pointer-to-the-character and the null terminator. We need to compare the pointed-to-character with the null terminator. To do this, we need to dereference the pointer. Change the line to this:
          ```
          while (*password != '\0') {
          ```
    - Recompile your code. You can now see that this warning does not appear and there are three warnings left.

3. Fix the remaining compiler warnings in pwd_checker.c

Part 2: Assert Statements

1. Learn about how you can use assert statements to debug your code. Edit the code in pwd_checker according to the directions in this section.
    - Compile and run your code

      ```
      gcc pwd_checker.c test_pwd_checker.c -o pwd_checker
      ```

      ```
      ./pwd_checker
      ```

    - The program says that qrtv?,mp!ltrA0b13rab4ham is not a valid password for Abraham Garcia. However, we can see that this password fits all of the requirements. It looks like there is a bug in our code.
    - The function check_password makes several function calls to verify that the password meets each of the requirements. To find the location of our bug, we can use assert statements to figure out which function is not returning the expected value. For example, you can add the following line after the function call to check_lower to verify that the function returns the correct value. We expect check_lower to return true because the password contains a lower case letter.

      ```
      assert(lower)
      ```

2. Add the remaining assert statements after each function call in the function check_password. This will help you determine which functions are not working as expected.
3. Compile your code.
4. Oh no! We just created a new compiler warning! Learn how to fix this warning using the man pages
    - The warning states that we have an implicit declaration of the function assert. This means that we do not have a definition for assert. This often means that you have forgotten to include a header file or that we spelled something wrong. The only thing that we have added since the last time our code was compiling without warnings is assert statements. This must mean that we need to include the definition of the function assert. assert is a library macro, so we can use the man pages to figure out which header file we need to include. Type the following into your terminal to pull up the man pages

      ```
      man assert
      ```

- The synopsis section tells us which header file to include. We can see that in order to use assert we need to include assert.h
- Add the following line to the top of pwd_checker.c
  ```
  #include <assert.h>
  ```
  Note that it is best practice to put your include statements in alphabetical order to make it easier for someone else reading your code to see which header files you have included. System header files should come before your own header files.
- Compile your code. There should be no warnings.
- Run your code. We can see that the assertion length failed. Look back at the function check_password. It looks like the function check_lower is working properly because assert(length) comes after assert(lower). This failed assertion is telling us that check_length is not working properly for this test case. We will investigate this in Part 3.

Part 3: Intro to GDB: start, step, next, finish, print, quit

What is GDB?

Here is an exerpt from the GDB website:

GDB, the GNU Project debugger, allows you to see what is going on 'inside' another program while it executes -- or what another 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:

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

In this class, we will be using cgdb which provides a lightweight interface to gdb to make it easier to use. CGDB is already installed on the Hive machines, so there is no installation required. The remainder of the document uses cgdb and gdb interchangeably.

GDB reference card

1. In this section, you will learn the GDB commands start, step, next, finish, print, and quit. This section will resolve bug(s) along the way. Make sure to fix the bug(s) in the code before moving on.
    - Before we can run our code through GDB, we need to include some additional debugging information in the executable. To do this, you will compile your code with the -g flag
      ```
      gcc -g pwd_checker.c test_pwd_checker.c -o pwd_checker
      ```
    - To start cgdb, run the following command. Note that you should be using the executable (pwd_checker) as the argument, not the source file (pwd_checker.c)
      ```
      cgdb pwd_checker
      ```
    - You should now see cgdb open. The top window displays our code and the bottom window diplays the console
    - Start running your program at the first line in main by typing the following command into the console. This will set a breakpoint at line 1 and begin running the program.
      ```
      start
      ```
    - The first line in main is a call to printf. We do not want to step into this function. To step over it, you can use the following command
      ```
      next
      ```
      or
      ```
      n
      ```
    - Step into check_password.
      ```
      step
      ```
      or
      ```
      s
      ```
    - Step into check_lower.
    - We have already seen that check_lower behaves properly with the given test case, so there is nothing that we need to look at here. To get out of this function, type the following command into the console
      ```
      finish
      ```
      Alternatively, you could have stepped until you reached the end of the function and it would have returned.
    - Step to the next line. We do not want to step into the assert function call because this is a library function, so we know that our error isn't going to be in there. Step over this line.
    - Step into check_length.
    - Step over strlen because this is a library function.
    - Step to the last line of the function.

- Let's print out the return value
  ```
  print meets_len_req
  ```
  or
  ```
  p meets_len_req
  ```
- Hmmm... it's false. That's odd. Let's print out length.
- The value of length looks correct, so there must be some logic error on line 24
- Ahah, the code is checking if length is less than or equal to 10, not greater than or equal. Update this line to
  ```
  bool meets_len_req = (length >= 10);
  ```
- Let's run our code to see if this works. First, we need to quit out of gdb which you can do with the following commands
  ```
  quit
  ```
  or
  ```
  q
  ```
- GDB will ask you to make sure that you want to quit. Type
  ```
  y
  ```
- Compile and run your code.
- Yay, it worked! We can see that we are failing the assertion name meaning that check_name is not behaving as expected.

2. Debug check_name on your own using the commands you just learned. Make sure you use the -g flag when compiling. Make sure that you are using cgdb to debug so that you practice using it because it will help you with your projects.
3. check_number is now failing, we will address this in the next part.

Part 4: Intro to GDB: break, conditional break, run, continue

1. In this section, you will learn the gdb commands break, conditional break, run, and continue. This section will resolve bug(s) along the way. Make sure to fix the bug(s) in the code before moving on.
   - Recompile and run your code. You should see that the assertion number is failing
   - Start cgdb
     ```
     cgdb pwd_checker
     ```
   - Let's set a breakpoint in our code to jump straight to the the function check_number using the following command
     ```
     break pwd_checker.c:check_number
     ```
     or
     ```
     b pwd_checker.c:check_number
     ```

- Use the following command to beginning running the program
  ```
  run
  ```
  or
  ```
  r
  ```
- Your code should run until it gets to the breakpoint that we just set.
- Step into check_range.
- Recall that the numbers do not appear until later in the password. Instead of stepping through all of the non-numerical characters at the beginning of password, we can jump straight to the point in the code where the numbers are being compared using a conditional breakpoint. A conditional breakpoint will only stop the program based on a given condition. The first number in the password 1, so we can set the breakpoint when letter is '1'. To set this breakpoint, enter the following line
  ```
  b 31 if letter=='1'
  ```
  We are using the single quote because 1 is a char. We did not need to specify the file name as we did last time, because we are already in pwd_checker.c.
- To continue executing your code after it stops at a breakpoint, use the following command
  ```
  continue
  ```
  or
  ```
  c
  ```
- The code has stopped at the conditional breakpoint. To verify this, print letter
  ```
  p letter
  ```
- It printed 49 '1' which is a decimal number followed by it's corresponding ASCII representation. If you look at an ASCII table, you can see that 49 is the decimal representation of the character 1.
- Let's take a look at the return value of check_range. Print is_in_range.
- The result is false. That's strange. '1' should be in the range.
- Let's look at the upper and lower bounds of the range. Print lower and upper.
- Ahah! The ASCII representation of lower is \000(the null terminator) and the ASCII representation of upper is \t. It looks like we passed in the numbers 0 and 9 instead of the characters '0' and '9'! Let's fix that
  ```
  if (check_range(*password, '0', '9')) {
  ```
- Quit cgdb and compile and run your code.

2. Debug check_upper on your own using the commands you just learned.

Exercise 3

Learning Goals:

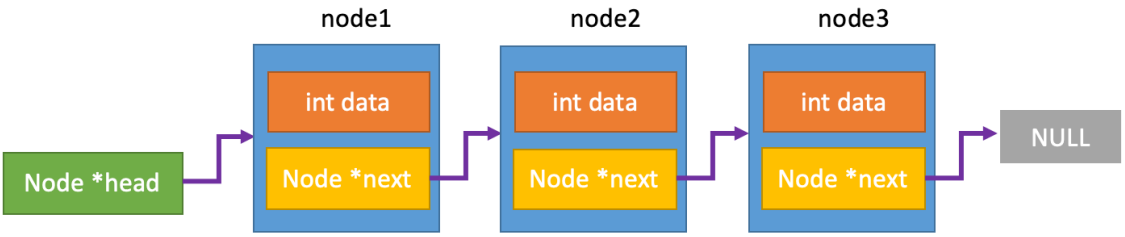
1. See an application of double pointers
2. Get more practice using gdb
3. Learn how to find the location of segfaults using gdb

Part 1: Double Pointers

1. When do we use double pointers?
   Here is a scenario to help you understand when to use double pointers.
   a. Let's say that we have the following struct:

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;
```

And the following linked list:



We want to have a function called update_integer that will update the value of an integer. It's signature should look like:

```
void update_integer(SOMETYPE int_to_update)
```

   b. What should be the type of int_to_update?

int *

We must use a pointer of to the integer in order to change the value of the original integer. If we just pass int, we will only change the local copy of the integer, and the change will not be reflected outside of the function.

c. How would we call update_integer to update the data field of node1?

   update_integer(&(head->data))
   Just as before, we must pass in a pointer to the integer in order to change the value of the original integer.

d. We want to have a function called update_head that will update the head of a list. It's signature should look like:

```
void update_head(SOMETYPE head_to_update)
```

e. What should be the type of head_to_update?

   Node **
   Just as before, whenever we want to update the value of a variable, we need to pass a pointer to it. In this scenario, the variable that we want to update is a pointer, so we need to pass a pointer to a pointer.

f. How would we call update_head to ensure that head is updated properly?

   update_head(&head)
   Again, we need to pass the pointer to the head to ensure that the original head is being updated and not just the local copy.

2. Read over the function add_to_front in exercise3/linked_list.c to see an example of double pointers.

Part 2: Printing

In GDB, you can print out more than just simple variables. Here are some examples:

You can print out a member of a struct

```
p my_struct->my_member
```

You can print out the result of a function

```
p strlen(my_string)
```

You can print the dereferenced value of a pointer

```
p *my_pointer
```

Part 3: How to find segfaults

In this section, we will be debugging the recursive implementation of reversing a linked list.

1. Read over the code in exercise3/linked_list.c.
2. In this step, you will learn how to find **segfaults** using cgdb. This section will resolve bug(s) along the way. Make sure to fix the bug(s) in the code before moving on.
    a. Compile and run your code.
    b. We can see that there is a segfault, but we do not know where it is occuring.
    c. Let's open up the code in cgdb (make sure that you compiled it with the -g flag).
    d. Recall the run command will run your code until something stops it (like a segfault or a breakpoint). Execute the run command.
    e. The command window shows us that the the program encountered a segfault ("Program received signal SIGSEGV") at line 62. Now we know the line where the segfault occured.
    f. Another command that can be useful when you are debugging segfaults is backtrace. This will allow you to see the call stack at the time of the crash. Execute the following command to see the call stack

    ```
    backtrace
    ```

    or

    ```
    bt
    ```

    In this case, we can see that reverse_list was called from main. We already knew this, so backtrace was not helpful in this case, but it can be helpful in the future if you have longer call stacks.

    g. Let's examine the state of the program when the segfualt occured by printing some variables. Print out head.
    h. Remember that head is a double pointer, so we expect to see an address when we print it out. head looks fine. Let's print out the dereferenced head (*head).
    i. It looks like *head is NULL. If you look at the line of code that the program is segfaulting on, you can see that we are trying to access (*head)->next. We know that trying to access a member of a NULL struct will result in a segfault, so this is our problem.

j. What does it mean for *head to be NULL? It means that the list is empty. How do we reverse an empty list? The reverse of an empty list is just an empty list, so we do not need to modify the list. At the top of the function, we can add a check to see if the list if empty and return if so. Modify the first line of the function to be the following:

```
if (head == NULL || *head == NULL) {
```

k. Compile and run your code. It should pass the reverse list tests now.

3. Debug add_to_back on your own using the commands you just learned. Make sure that you are using cgdb so that you can get practice using it. This will help you with your projects.

Exercise 4

Here's one to help you in your interviews. In ll_cycle.c, complete the function ll_has_cycle() to implement the following algorithm for checking if a singly- linked list has a cycle.

1. Start with two pointers at the head of the list. One will be called fast_ptr and the other will be called slow_ptr.
2. Advance fast_ptr by two nodes. If this is not possible because of a null pointer, we have found the end of the list, and therefore the list is acyclic.
3. Advance slow_ptr by one node. (A null pointer check is unnecessary. Why?)
4. If the fast_ptr and slow_ptr ever point to the same node, the list is cyclic. Otherwise, go back to step 2.

If you want to see the definition of the node struct, open ll_cycle.h

Action Item

Implement ll_has_cycle(). Once you've done so, you can execute the following commands to run the tests for your code. If you make any changes, make sure to run ALL of the following commands again, in order.

```
gcc -g -o test_ll_cycle test_ll_cycle.c ll_cycle.c
./test_ll_cycle
```

Here's a Wikipedia article on the algorithm and why it works. Don't worry about it if you don't completely understand it. We won't test you on this.

Tasks

- Make sure to learn all compiling and running commands
- Complete all the codes and rectify errors
- Show your output