Lab 2: C Debugging

Deadline: Thursday, September 7, 11:59:59 PM PT

For this lab, please complete the exercises in the order listed. The exercises may depend on each other.

Lab Slides

Setup

You must complete this lab on the hive machines. See Lab 0 for a refresher on using them.

In your `labs` directory, pull the files for this lab with:

```
$ git pull starter main
```

If you get an error like the following:

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

make sure to set the starter remote as follows:

```
$ git remote add starter https://github.com/61c-teach/fa23-lab-starter.git
```

and run the original command again.

If you run into any `git` errors, please check out the common errors page.

Exercise 1: Compiler Warnings and Errors

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.

1. **Read** over the code in `ex1_compiler_warnings.c`.
2. **Compile** your program with `gcc -o ex1_compiler_warnings ex1_compiler_warnings.c`. You should see 3 warnings.

3. **Read** the first line of the first warning. The line begins with `ex1_compiler_warnings.c:13:22`, which tells you that the warning is caused by line 13 of `ex1_compiler_warnings.c`. The warning states that the program is trying to assign a `char` to a `char *`.
4. **Open** `ex1_compiler_warnings.c` and **navigate** to the line that's causing the warning. It is trying to assign a `char` to a `char *`. The compiler has pointed this out as a potential error because we should not be assigning a `char` to a `char *`.
5. **Fix** this compiler warning.
6. **Recompile** your code. You can now see that this warning does not appear anymore and there are 2 warnings left.
7. **Fix** the remaining compiler warnings in `ex1_compiler_warnings.c`.

What is GDB?

Here is an excerpt 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:

- 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.

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 this class uses CGDB and GDB interchangeably.

Here's a GDB reference card .

If you run into any issues with GDB, see the Common GDB Errors section below

Exercise 2: Intro to GDB

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.

The table below is a summary of the above commands

Command	Abbreviation	Description
start	N/A	begin running the program and stop at line 1 in main
step	s	execute the current line of code (this command will step into functions)
next	n	execute the current line of code (this command will not step into functions)
finish	fin	executes the remainder of the current function and returns to the calling function
print [arg]	p	prints the value of the argument
quit	q	exits gdb

You should be filling in `ex2_commands.txt` with the corresponding commands. Please only use the commands from the table above. **For correctness, we will be checking the output of your `ex2_commands.txt` against a desired output.** We'd recommend opening two SSH windows so you can have the commands file and the `cgdb` session at the same time. Even though you are adding to `ex2_commands.txt`, please check your work by actually running these commands in `cgdb`.

1. **Compile** your program with the `-g` flag. This will include additional debugging information in the executable that CGDB needs.

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

2. **Start** `cgdb`. 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 displays the console.

For each of the following steps, add the CGDB commands you execute to `ex2_commands.txt`. Each command should be on its own line. Each step below will require one or more CGDB commands.

1. **Start** your program the first line in `main`. Hint: this command should set a breakpoint at line 1 and begin running the program.
2. The first line in `main` is a call to `printf`. We do not want to step into this function. **Step over** this line in the program.
3. **Step until** the program is on the `check_password` call. Note that the line with an arrow next to it is the line we're currently on, but has not been executed yet.
4. **Step into** `check_password`.

5. **Step into** `check_lower`.
6. **Print** the value of `password` (`password` is a string).
7. **Step out** of `check_lower` immediately. Do not step until the function returns.
8. **Step into** `check_length`.
9. **Step to** the last line of the function.
10. **Print** the return value of the function. The return value should be `false`.
11. **Print** the value of `length`. It looks like `length` was correct, so there must be some logic issue on line 24.
12. **Quit** CGDB. CGDB might ask you if you want to quit, type `y` (but do not add `y` to `ex2_commands.txt`).

At this point, your `ex2_commands.txt` should contain a list of commands from the steps above. You don't need to add anything from the steps below to your `ex2_commands.txt`.

1. **Fix** the bug on line 24.
2. **Compile** and **run** your code.
3. The program still fails. Open and step through `cgdb` again, you should see that `check_number` is now failing. We will address this in the next exercise.

Exercise 3: More GDB

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.

The table below is a summary of the above commands

Command	**Abbreviation**	**Description**
break [line num or function name]	b	set a breakpoint at the specified location, use `filename.c:linenum` to set a breakpoint in a specific file
conditional break (ex: break 3 if n==4)	(ex: b 3 if n==4)	set a breakpoint at the specified location only if a given condition is met
run	r	execute the program until termination or reaching a breakpoint
continue	c	continues the execution of a program that was paused

You should be filling in `ex3_commands.txt` with the corresponding commands. Please only use the commands from the table above and the table for exercise 2. **For correctness, we**

will be checking the output of your `ex3_commands.txt` against a desired output. We'd recommend opening two SSH windows so you can have the commands file and the `cgdb` session at the same time. Even though you are adding to `ex3_commands.txt`, please check your work by actually running these commands in `cgdb`.

1. **Recompile and run** your code. You should see that the assertion `number` is failing
2. **Start** cgdb

```
$ cgdb pwd_checker
```

For each of the following steps, add the CGDB commands you execute to `ex3_commands.txt`. Each command should be on its own line. Each step below will require one or more CGDB commands.

1. **Set a breakpoint** in our code to jump straight into the the function `check_number`. Your breakpoint should not be in `check_password`.
2. **Run** the program. Your code should run until it gets to the breakpoint that we just set.
3. **Step into** `check_range`.
4. 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 `0`, so we can set the breakpoint when `letter` is `'0'`. **Break on line 31 if the `letter` is `'0'`**.

   We are using the single quote because `0` is a char.
5. **Continue executing** your code after it stops at a breakpoint.
6. The code has stopped at the conditional breakpoint. To verify this, **print** `letter`.

   It should print `48 '0'` which is a decimal number followed by it's corresponding ASCII representation. If you look at an [ASCII table](), you can see that `48` is the decimal representation of the character `0`.
7. Let's take a look at the return value of `check_range`. **Print** `is_in_range`. The result is `false`. That's strange. `'0'` should be in the range.
8. Let's look at the upper and lower bounds of the range. **Print** `lower`.
9. **Print** `upper`.
10. 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'`!
11. **Quit** CGDB. CGDB might ask you if you want to quit, type `y` (but do not add `y` to `ex3_commands.txt`).

At this point, your `ex3_commands.txt` should contain a list of commands from the steps above. You don't need to add anything from the steps below to your `ex3_commands.txt`.

1. **Fix** the bug.
2. **Compile** and **run** your code. There's one more error, which you will find in exercise 4.

Exercise 4: Debug

1. **Debug** `check_upper` on your own using the commands you just learned. The function appears to be returning `false` even though there's an uppercase letter. Hint: the bug itself may not be in `check_upper` itself.

Valgrind

Even with a debugger, we might not be able to catch all bugs. Some bugs are what we refer to as "bohrbugs", meaning they manifest reliably under a well-defined, but possibly unknown, set of conditions. Other bugs are what we call "heisenbugs", and instead of being determinant, they're known to disappear or alter their behavior when one attempts to study them. We can detect the first kind with debuggers, but the second kind may slip under our radar because they're (at least in C) often due to mis-managed memory. Remember that unlike other programming languages, C requires you (the programmer) to manually manage your memory.

We can use a tool called Valgrind to help catch to help catch "heisenbugs" and "bohrbugs". Valgrind is a program which emulates your CPU and tracks your memory accesses. This slows down the process you're running (which is why we don't, for example, always run all executables inside Valgrind) but also can expose bugs that may only display visible incorrect behavior under a unique set of circumstances.

Let's take a look at the `bork` translation program! Bork is an ancient language that is very similar to English. To translate a word to Bork, you take the English word and add an 'f' after every vowel in the word.

Let's see if we can understand some Bork. Compile and run `bork` using the following commands.

```
$ gcc -g -o bork bork.c
$ ./bork hello
```

An example output is provided below. Note that your output will probably look different.

```
Input string: "hello"
Length of translated string: 21
Translate to Bork: "hefl2?^?Ul2?^?Uof?^?U"
```

Hmm, Bork is an old language, but there shouldn't be all of these strange characters. It seems that perhaps the ancients left some bugs in their program! Shall we embark on a journey to squash bugs and uncover the true beauty of Bork?

If we take a brief glance at `main`, we can see that we are taking an input string (`src_str`) and translating it to Bork (`dest_str`). If we scroll to the top, we can see that we have a function (`alloc_str`) to allocate space for a string in the heap, a `Str` struct which contains a string and it's length, a `make_str` function which will create a `Str` struct and initialize its `data` and `len` field, and a function to free our struct's data. There is also a function to concate two strings together and another function to translate a letter to Bork. Now this is quite a long program to debug.

Wouldn't it be nice if there were a tool that gave us a good first place to look?

Well as it turns out, there are a couple and `valgrind` is one of them!

Let's run `valgrind` on our program using the following command.

```
$ valgrind ./bork hello
```

```
==10170== $ Memcheck, a memory error detector
==10170== $ Copyright (C) $ 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10170== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10170== Command: ./bork hello
==10170==
==10170== Invalid read of size 1
==10170==    at 0x4C34D04: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==10170==    by 0x10879F: make_Str (bork.c:22)
==10170==    by 0x108978: translate_to_bork (bork.c:56)
==10170==    by 0x1089F2: main (bork.c:68)
==10170==  Address 0x522f041 is 0 bytes after a block of size 1 alloc'd
==10170==    $ at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.
==10170==    $ by 0x108781: alloc_str (bork.c:10)
==10170==    $ by 0x10895E: translate_to_bork (bork.c:54)
==10170==    $ by 0x1089F2: main (bork.c:68)
==10170==
==10170== $ Invalid read of size 1
==10170==    $ at 0x4C34D04: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.
==10170==    $ by 0x10879F: make_Str (bork.c:22)
==10170==    $ by 0x108952: translate_to_bork (bork.c:51)
==10170==    $ by 0x1089F2: main (bork.c:68)
==10170==  $ Address 0x522f0e2 is 0 bytes after a block of size 2 alloc'd
==10170==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==10170==    by 0x108781: alloc_str (bork.c:10)
==10170==    by 0x10892D: translate_to_bork (bork.c:48)
==10170==    by 0x1089F2: main (bork.c:68)
==10170==
Input string: "hello"
Length of translated string: 7
==10170== Invalid read of size 1
==10170==    at 0x4C34D04: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==10170==    by 0x4E9B4A2: vfprintf (vfprintf.c:1643)
```

```
==10170==    by 0x4EA2EE5: printf (printf.c:33)
==10170==    by 0x108A6F: main (bork.c:74)
==10170==  Address 0x522f317 is 0 bytes after a block of size 7 alloc'd
==10170==    $ at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.
==10170==    $ by 0x108781: alloc_str (bork.c:10)
==10170==    $ by 0x108833: concat (bork.c:32)
==10170==    $ by 0x108A15: main (bork.c:69)
==10170==
$ Translate to Bork: "hefllof"
==10170==
==10170== $ HEAP SUMMARY:
==10170==     $ in use at exit: 7 bytes in 1 blocks
==10170==   $ total heap usage: 11 allocs, 10 frees, 1,051 bytes allocated
==10170==
==10170== $ LEAK SUMMARY:
==10170==    $ definitely lost: 7 bytes in 1 blocks
==10170==    $ indirectly lost: 0 bytes in 0 blocks
==10170==      $ possibly lost: 0 bytes in 0 blocks
==10170==    $ still reachable: 0 bytes in 0 blocks
==10170==         $ suppressed: 0 bytes in 0 blocks
==10170== $ Rerun with --leak-check=full to see details of leaked memory
==10170==
==10170== $ For counts of detected and suppressed errors, rerun with: -v
==10170== $ ERROR SUMMARY: 6 errors from 3 contexts (suppressed: 0 from 0)
```

(Interesting side note: when we look at the normal program output in this valgrind log, we see normal behavior (i.e. it prints "hefllof"). That's because the way valgrind runs our program is different than how our program runs "naturally" (aka "bare metal"). We're not going to get into that for now.)

But back on debugging: A good general rule of thumb to follow when parsing big error logs is to only consider the first error message (and ignore the rest), so let's do that:

```
==10170== Invalid read of size 1
==10170==    at 0x4C34D04: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==10170==    by 0x10879F: make_Str (bork.c:22)
==10170==    by 0x108978: translate_to_bork (bork.c:56)
==10170==    by 0x1089F2: main (bork.c:68)
```

The error message states that we are doing an invalid read of size 1. What does this mean? An invalid read means that your program is reading memory at a place that it shouldn't be (this can cause a segfault, but not always). Size 1 means that we were attempting to read 1 byte.

Because we're unfamiliar with this ancient codebase and we don't want to read all of it to find the bug, a good process to follow is to start at high-level details and work our way down (so basically work our way through the call stack that valgrind provides).

Let's look at bork.c line 68 in `main` (the botton of the stack):

```
Str bork_substr = translate_to_bork(src_str.data[i]);
```

Is something funky going on here? Looks like we are just passing a character to `translate_to_bork`. Seems ok so far.

Let's go farther down the call stack and look at bork.c line 56 in `translate_to_bork`:

```
return make_Str(res);
```

We're just calling `make_Str` here. We should go deeper. Let's look at bork.c line 22.

```
return (Str){.data=str,.len=strlen(str)};
```

Here we are making a new `Str` struct and setting its `data` and `len` parameters. That seems normal too!

But `valgrind` says that `strlen` is doing an invalid read?

Well, we're passing a string to it right? What does `strlen` do again? It determines the length of a string by iterating over each character until it gets to a null terminator. Maybe there is no null terminator so `strlen` keeps going past the end of the string (which would mean that it's going past the area that we allocated for the string).

Let's make sure our string has a null terminator by checking where we created it.

Earlier, we saw this on line 56 in `translate_to_bork`.

```
return make_Str(res);
```

If we look two lines up (line 54), we can see that we are allocating space for the string by calling `alloc_str`. Let's take a look at this function.

```
char *alloc_str(int len) {
    return malloc(len*sizeof(char));
}
```

Hmmm. It looks like `alloc_str` is giving us some memory that's only `len` big, which means when we write to the string in `translate_to_bork`, we don't have enough space for a null terminator!

Let's make the following change to fix the problem:

```
10c10,12
<     return malloc(len*sizeof(char));
```

```
---
>     char *data = malloc((len+1)*sizeof(char));
>     data[len] = '\0';
>     return data;
```

Let's run our program to see if we fixed the problem

```
$ ./bork hello
```

```
Input string: "hello"
Length of translated string: 7
Translate to Bork: "hefllof"
```

Everything looks like it's working properly. However, there could be hidden errors that we cannot see, so let's run our code through valgrind to make sure that there are no underlying issues.

```
$ valgrind ./bork hello
```

```
==29797== Memcheck, a memory error detector
==29797== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29797== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==29797== Command: ./bork hello
==29797==
Input string: "hello"
Length of translated string: 7
Translate to Bork: "hefllof"
==29797==
==29797== HEAP SUMMARY:
==29797==     in use at exit: 8 bytes in 1 blocks
==29797==   total heap usage: 11 allocs, 10 frees, 1,061 bytes allocated
==29797==
==29797== LEAK SUMMARY:
==29797==    definitely lost: 8 bytes in 1 blocks
==29797==    indirectly lost: 0 bytes in 0 blocks
==29797==      possibly lost: 0 bytes in 0 blocks
==29797==    still reachable: 0 bytes in 0 blocks
==29797==         suppressed: 0 bytes in 0 blocks
==29797== Rerun with --leak-check=full to see details of leaked memory
==29797==
==29797== For counts of detected and suppressed errors, rerun with: -v
==29797== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Let's take a look at the heap summary below. It tells us that we had 8 bytes in 1 block allocated at the time of exit. This means that the memory in the heap that was not free'd stems from one allocation call and that it is 8 bytes large.

Next, we can see the heap summary which shows that we made 11 allocation calls and 10 frees over the lifetime of the program.

```
==29797== HEAP SUMMARY:
==29797==     in use at exit: 8 bytes in 1 blocks
==29797==   total heap usage: 11 allocs, 10 frees, 1,061 bytes allocated
```

Now let's take a look at the leak summary below. This just states that we lost 8 bytes in 1 block.

```
==29797== LEAK SUMMARY:
==29797==    definitely lost: 8 bytes in 1 blocks
==29797==    indirectly lost: 0 bytes in 0 blocks
==29797==      possibly lost: 0 bytes in 0 blocks
==29797==    still reachable: 0 bytes in 0 blocks
==29797==         suppressed: 0 bytes in 0 blocks
==29797== Rerun with --leak-check=full to see details of leaked memory
```

It tells us to "Rerun with --leak-check=full to see details of leaked memory", so let's do that.

```
$ valgrind --leak-check=full ./bork hello
```

```
==32334== Memcheck, a memory error detector
==32334== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==32334== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==32334== Command: ./bork hello
==32334==
Input string: "hello"
Length of translated string: 7
Translate to Bork: "hefllof"
==32334==
==32334== HEAP SUMMARY:
==32334==     in use at exit: 8 bytes in 1 blocks
==32334==   total heap usage: 11 allocs, 10 frees, 1,061 bytes allocated
==32334==
==32334== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==32334==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==32334==    by 0x108784: alloc_str (in /home/cc/cs61c/fa22/staff/cs61c-tac/bork)
==32334==    by 0x10884E: concat (in /home/cc/cs61c/fa22/staff/cs61c-tac/bork)
==32334==    by 0x108A30: main (in /home/cc/cs61c/fa22/staff/cs61c-tac/bork)
==32334==
==32334== LEAK SUMMARY:
==32334==    definitely lost: 8 bytes in 1 blocks
==32334==    indirectly lost: 0 bytes in 0 blocks
==32334==      possibly lost: 0 bytes in 0 blocks
==32334==    still reachable: 0 bytes in 0 blocks
==32334==         suppressed: 0 bytes in 0 blocks
```

```
==32334==
==32334== For counts of detected and suppressed errors, rerun with: -v
==32334== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Now Valgrind is telling us the location where the unfree'd block was initially allocated. Let's take a look at this below. If we follow the call stack, we can see that `malloc` was called by `alloc_str` which was called by `concat` in `main`.

```
==32334== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==32334==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==32334==    by 0x108784: alloc_str (in /home/cc/cs61c/fa22/staff/cs61c-tac/bork)
==32334==    by 0x10884E: concat (in /home/cc/cs61c/fa22/staff/cs61c-tac/bork)
==32334==    by 0x108A30: main (in /home/cc/cs61c/fa22/staff/cs61c-tac/bork)
```

If we look in `main`, we can see that we allocate the space for `dest_str` by calling `concat`, but we never free it. We need `dest_str` until the end of the program, so let's free it right before we return from `main`. This struct was allocated on the stack in main (`Str dest_str={};`), so we do not need to free the struct itself. However, the data that the struct points to was allocated in the heap. Therefore, we only need to free this portion of the struct. If you take a look near the top of the program, we have already provided a function `free_Str` to free the allocated portion of the struct. Let's call this function at the end of our program.

```
76a77
>     free_Str(dest_str);
```

You might be wondering why we are not freeing `src_str`. If we take a look at where we constructed `src_str` (`Str src_str = make_Str(argv[1]);`), we can see that it was created using `make_str` which does not make any calls to allocate space on the heap. The string that we are using to make `src_str` comes from `argv[1]`. The program that calls main is responsible for setting up `argv[1]`, so we don't have to worry about it.

Once we fix our error, the valgrind output should look like this. The heap summary shows that there are no blocks allocated at the time we exit. The error summary at the bottom shows us that there are no errors to report.

```
$ valgrind ./bork hello
```

```
==10835== Memcheck, a memory error detector
==10835== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10835== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10835== Command: ./bork hello
==10835==
Input string: "hello"
Length of translated string: 7
```

```
Translate to Bork: "hefllof"
==10835==
==10835== HEAP SUMMARY:
==10835==     in use at exit: 0 bytes in 0 blocks
==10835==   total heap usage: 11 allocs, 11 frees, 1,061 bytes allocated
==10835==
==10835== All heap blocks were freed -- no leaks are possible
==10835==
==10835== For counts of detected and suppressed errors, rerun with: -v
==10835== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exercise 5: Using Valgrind to find segfaults

There's a bug in `ex5_valgrind`, let's see how we can detect it with valgrind.

1. **Compile** `ex5_valgrind.c`. Notice that there are no compiler errors or warnings, and we're using the `-g` flag in case we need to debug this program in the future.
   ```
   $ gcc -g -o ex5_valgrind ex5_valgrind.c
   ```
2. **Run** `ex5_valgrind`. Notice that the program doesn't throw any errors.
3. **Run** `valgrind` on `ex5_valgrind`. You should see that there are 2 errors.
4. **Read** the valgrind output carefully. In `ex5_answers.txt`, answer the following questions. Please don't change the formatting of the file. For question 1 through 7, we are referring to the first `valgrind` error (an invalid write error).
   1. How many bytes are the invalid write? (The answer should be a number without any units)
   2. Which function caused the invalid write? (The answer should be the name of the function)
   3. Which function called the answer to question 2? (The answer should be the name of a function)
   4. Which file did the call occur in? (The answer should be the name of a file)
   5. Which line did the call occur on? (The answer should be a number)
   6. How many bytes were actually allocated? (The answer should be a number without any units)
   7. How many bytes should have been allocated? Feel free to read the code. (The answer should be a number without any units)
   8. Are there any memory leaks? (The answer should be Yes or No)
   9. How many bytes were leaked? Write 0 if there are no memory leaks. (The answer should be a number without any units)

Exercise 6: Memory Management

This exercise uses `ex6_vector.h`, `ex6_test_vector.c`, and `ex6_vector.c`, where we provide you with a framework for implementing a variable-length array. This exercise is designed to help familiarize you with C structs and memory management in C.

1. **Try to explain** why `bad_vector_new()` is bad. We have provided the reason here, so you can verify your understanding

   ▶ bad_vector_new()

2. **Fill in** the functions `vector_new()`, `vector_get()`, `vector_delete()`, and `vector_set()` in `ex6_vector.c` so that our test code `ex6_test_vector.c` runs without any memory management errors.

   Comments in the code describe how the functions should work. Look at the functions we've filled in to see how the data structures should be used. For consistency, *it is assumed that all entries in the vector are 0 unless set by the user. Keep this in mind as `malloc()` does not zero out the memory it allocates.* `vector_set` should resize the array if the index passed in is larger than the size of the array.
3. **Test** your implementation of `vector_new()`, `vector_get()`, `vector_delete()`, and `vector_set()` for correctness.

   ```
   $ gcc -g -o ex6_vector ex6_vector.c ex6_test_vector.c
   $ ./ex6_vector
   ```
4. **Test** your implementation of `vector_new()`, `vector_get()`, `vector_delete()`, and `vector_set()` for memory management.

   ```
   $ valgrind ./ex6_vector
   ```

Any number of suppressed errors is fine; they do not affect us.

Feel free to also use CGDB to debug your code.

Submission

Save, commit, and push your work, then submit to the **Lab 2** assignment on Gradescope.

Common GDB Errors

GDB is skipping over lines of code

This could mean that your source file is more recent than your executable. Exit GDB, recompile your code with the `-g` flag, and restart gdb.

GDB isn't loading my file

You might see an error like this "not in executable format: file format not recognized" or "No symbol table loaded. Use the "file" command."

This means that you called gdb on the source file (the one ending in `.c`) instead of the executable. Exit GDB and make sure that you call it with the executable.

How do I switch between the code window and the console?

CGDB presents a vim-like navigation interface: Press `i` on your keyboard to switch from the code window to the console. Press `Esc` to switch from the console to the code window.

GDB presents a readline/emacs-like navigation interface: Press `Ctrl` + `X` then `O` to switch between windows.

I'm stuck in the code window

Press `i` on your keyboard. This should get you back to the console.

The text UI is garbled

Refresh the GDB text UI by pressing `Ctrl` + `l`.

Other Useful GDB Commands (Recommended)

Command: `info locals`

Prints the value of all of the local variables in the current stack frame

Command: `command`

Executes a list of commands every time a break point is reached. For example:

Set a breakpoint:

```
b 73
```

Type `commands` followed by the breakpoint number:

```
commands 1
```

Type the list of commands that you want to execute separated by a new line. After your list of commands, type `end` and hit `Enter`.

```
p var1
p var2
end
```