Prepare your answers to the following questions in a single plain ASCII text file. If you work with a partner, make sure the submitted file contains a properly-completed copy of the partners form posted on the assignments page. Failure to do that will result in at least one of you not receiving credit for the assignment.

Submit your file to the Curator system, under the heading `c03`, by the posted deadline for this assignment. Late submissions will not be counted.

Download the associated tar file from the website, and unpack it into a subdirectory on your CentOS 7 installation or on rlogin. There will be separate subdirectories, containing code, for each question. You are advised to consult the course notes, the Matloff book online, and a good `gdb` cheat sheet. Note also that it is easy to copy text from a Linux terminal window and paste it into a text editor.

For each of the following questions, copy and paste the relevant part of your `gdb` session into your text file and explain your conclusions. In each part, you must copy the `gdb` command(s) you entered and the resulting, relevant `gdb` output. Since the point of this assignment is to learn to use some feature of `gdb`, answers without `gdb` verification will receive no credit.

1. [45 points] The file `q1.c` contains a short `main()` function, two helper functions, and finds real roots of quadratic equations. Compile `q1.c` with the following command:

```
gcc -o q1 -std=c11 -O0 -Wall -W -ggdb3 q1.c
```

Now we are going to explore some of the features of gdb.

a) Copy the last five lines of terminal output that are produced when you run `q1` as follows:

```
CentOS> q1 0.05 3.4 5.7 3.0
```

b) Start `gdb` on `q1`. Set a breakpoint on `main()`, and another breakpoint on line 57. Use the `gdb` command `info break` to display the breakpoints you just set; copy those commands and output.

c) Now, start the program with the parameters given in part **a)**, but redirect the program's output into a file named `q1output.txt`. The program should pause at line 43. Use the `gdb` command `continue` to resume execution, which should now pause at the second breakpoint at line 57. Copy those commands and output.

d) Now, use the `info locals` command to display the values of local variables. Copy those commands and output.

Most of the local variables were initialized from the command-line parameters used when you started `q1`. You will probably notice that the values of some, or all, of the local variables are not exactly what you expected. This is a side-effect of the inherent inaccuracies of working with the floating-point types.

e) Use the command `next`, several times, until `gdb` displays line 69. Be careful here, we do not want to execute line 69 yet. Now, use the `step` command to step into the call to the function `extrapolate()`. Note that `gdb` shows the values of the parameters that were passed to the function. `gdb` should now be displaying line 108 of the source code. Copy those commands and output.

f) Now use the command `next` to execute lines 108, 109 and 110, which set the local variables declared in the function `extrapolate()`. Use the command `info locals` again, and note this does not display the values of the parameters. Copy those commands and output.

g) Use the command `info args` to display the values of the parameters passed to `extrapolate()`. Copy those commands and output.

h) Use the command `next`, several times, until `gdb` displays line 75. Be careful here, we do not want to execute line 75 yet. Use the `print` command to display the value of the variable `funcval`. Copy those commands and output.

i) Use the command `next`, several times, until `gdb` displays line 66. Be careful here, we do not want to execute line 66 yet. Use the `print` command to display the value of the variable `h`. Copy those commands and output.

j) Use the command `next` until `gdb` displays line 69 again. Be careful here, we do not want to execute line 69 yet. Now, `step` into the call to `extrapolate()`, and then `step` into the call to `funcprime()`. Execute the command `backtrace`. This shows the chain of function calls that got you to this point. Copy those commands and output.

What you are seeing here is a simplified view of the *runtime stack*. Each time a function is called, the program creates an object called a *stack frame*; the stack frame is used to store information about local variables in the function, as well as other information (but not the code for the function). We will be studying the runtime stack in detail later in the course.

For now, note that you see there are three frames on the stack, since we got here by starting the function `main()`, which has called the function `extrapolate()`, which has called the function `funcprime()`.

k) Use the command `info frame 0`, which will display information about frame 0 on the stack. This is the "top" frame on the stack, which belongs to the last function called, `funcprime()`. Copy those commands and output.

What you are seeing here is a simplified view of the frame for `funcprime()`. There's not much interesting information here (yet), but you should notice the values of the parameters (args).

l) Just to be complete, use the commands `info frame 1` and `info frame 2` to display information about the frames for `extrapolate()` and `main()`, respectively. Copy those commands and output.

m) Repeat the command `next` until `gdb` displays line 78 again. Now, set a conditional breakpoint to pause execution at line 75 when the variable `i` reaches the value 250. Use the command `info break` again to show the current breakpoints. Copy those commands and output.

n) Use the `delete` command to remove the breakpoints you set at lines 43 and 57, then use `info break` to verify that you only have the last, conditional, breakpoint now. Copy those commands and output.

o) Use the `continue` command to resume execution; `gdb` should now pause at line 75. Display the values for the variables `h`, `approximation`, and `funcval`. Copy those commands and output.

Now use `continue` again, which should run the program until it terminates normally.

2. **[16 points]** The file `q2.c` contains a short `main()` function and the following recursive function:

```
int q2(int N) {

       if ( N <= 1 )
          return 1;
       else {
          int retVal = q2(--N);  // being clever (?) with decrement
          return N * retVal;
       }
}
```

Adapt the command given in question 1 to compile the given code for debugging; it compiles but does not always produce correct results:

```
centos > q2 5
5! = 24
centos > q2 3
3! = 2
```

Use `gdb` to discover any relevant clues to what goes wrong with the function above when you run the program with the parameter 3, as shown in the second example above. Describe your observations, and include supporting lines from your `gdb` output.

Note: your work here may not fully explain WHY things go wrong, but it should explain clearly WHAT goes wrong.

3. The file `q3.c` contains a short `main()` function and a second function, `q3()`, called repeatedly by `main()`:

```
int main(int argc, char** argv) {

   if ( argc != 2 ) {
      printf("Invocation:  q3 iterLimit, where iterLimit >= 0.\n");
      exit(1);
   }

   int iterLimit = atoi(argv[1]);  // convert argument to an integer
   if ( iterLimit < 0 ) {
      printf("iterLimit must be non-negative.\n");
      exit(2);
   }

   int value = -1;
   int pass = 1;
   while ( pass < iterLimit ) {

      value = q3(pass);
      pass++;
   }

   return 0;
}

int q3(int N) {

   static int dejavue = 1;    // dejavue retains its value for next call

   dejavue = N * dejavue + dejavue % 29;  // perform strange computation

   return ( dejavue % 500 );  // return value between 0 and 499, inclusive
}
```

a) **[13 points]** Use appropriate `gdb` commands to analyze the program and determine what value will be returned by the function `q3()` when it is called from the given loop with the parameter equaling 100. Show the relevant `gdb` output to support your answer.

b) **[13 points]** Use appropriate `gdb` commands to analyze the program and determine what value the variable `pass` will have at the beginning of the iteration on which `q3()` returns the value -146. Show the relevant `gdb` output to support your answer.

c) **[13 points]** Based on the information you discovered in part b), use appropriate `gdb` commands to analyze the program and determine what value the variable `dejavue` will have immediately before `q3()` returns the value -146. Show the relevant `gdb` output to support your answer.

Some Comments

If you are running a bash shell in your CentOS virtual machine, you can copy and paste text from the terminal window as you work. Just use your mouse to highlight the text you want to copy, then either enter Shift-Ctrl- C (hold all three keys down at once), or use the Copy option from the Edit menu. This places the selected text in the system clipboard. You can then just use Ctrl-C in your text editor (e.g., geany) to paste the text.

An alternative, especially if you are using your rlogin account, is to use the `script` command. Briefly, if you enter the command `script` in the bash shell, all output that subsequently appears in that terminal window will also be logged to a file, named `typescript`. Once you have started the logging, just run your `gdb` session normally.

Entering Ctrl-D causes the logging of terminal output to stop. Unfortunately, the file usually contains quite a few non-ASCII characters, and the file may be difficult to read. That can be remedied by running the following command:

```
cat typescript | perl -pe 's/\e([^\[\]]|\[.*?[a-zA-Z]|\].*?\a)//g' | col -b > log.txt
```

This will create a cleaned-up version of the `script` command's log file. There is a bash shell script, `dejunk.sh`, posted on the course website that encapsulates the command above. You can just download that script to your CentOS account, use `chmod` to set execute privileges for the file, and use the command below to generate a cleaned-up version:

```
centos > ./dejunk.sh typescript log.txt
```

Either way, this will create a complete log of your `gdb` session, and you can extract the parts you need from that.

Change Log

Any changes or corrections to the specification will be documented here.

Version	Posted	Pg	Change
1.00	Aug 26		Base document
1.01	Sept 12	2	Added reference to the build command given in Q1 to the instructions for Q2