> If debugging is the process of removing bugs, then programming must be the process of putting them in.
>
> *Edsger W. Dijkstra*

Prepare your answers to the following questions in a single plain ASCII text file. If you work with a partner, only one of you should make a submission, and make sure the submitted file contains a properly-completed copy of the partners form posted on the assignments page. Failure to do that may 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 unless an extension has been approved by a course instructor.

Download the associated tar file from the website, and unpack it into a subdirectory on your CentOS 8 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 at the Safari database via the VT Library), 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 (or your terminal commands) into your text file and explain your conclusions. In each part, you must copy the `gdb` or terminal command(s) you entered and the resulting, relevant `gdb` or terminal output. Since the point of this assignment is to learn to use some feature of `gdb`, answers without `gdb` verification will receive no credit.

One warning: when stepping through the code with `next` or `step`, it's easy to halt at the wrong instruction. Remember that the last source code statement shown has not been executed yet.

1. [30 points] The file `q1.c` contains a short `main()` function that uses typecasts to illustrate some of the hazards of comparing signed and unsigned integers. Compile `q1.c` with the following command[1]:

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

This will result in some warnings. Some are about comparing integers with different signedness; that's actually one point of this question, so we'll disregard those. The other warnings are about unused variables; that's due to the fact that we declare and set some variables, so their values can be checked in `gdb`, but we don't use those values in the code. Those warnings are OK as well. Now we are going to explore some of the features of `gdb`.

a) Begin a `gdb` session on `q1`, and set a breakpoint at line 24:

```
uint32_t LT0 = 0;
if ( X < Y ) {        // line 24
   LT0 = 1;           // line 25
}
```

Run `q1` with the parameters -200 and 500. What value does `LT0` have after the if-statement completes execution? Is that what you should have expected, given the values of `X` and `Y`? (Determining this will require using `gdb` commands `next` and `print`.)

b) Was line 25 executed or skipped over?

c) Now use the command `next` to complete the execution of the next if-statement:

```
uint32_t LT1 = 0;
if ( (uint32_t) X < Y ) {
   LT1 = 1;
}
```

What value does `LT1` have after the if-statement completes execution?

d) Use `gdb` to display the value of `(uint32_t) X`. Does this explain the value given to `LT1`?

e) Now use the command next to complete the execution of the next if-statement:

```
uint32_t LT2 = 0;
if ( X < (uint32_t) Y ) {
   LT2 = 1;
}
```

What value does `LT2` have after the if-statement completes execution? What does that suggest about the way C handles typecasts in mixed-sign expressions?

f) Now use the command next to complete the execution of the final if-statement:

```
uint32_t GT1 = 1;
if ( (uint32_t) Y > X ) {
   GT1 = 0;
}
```

What value does `GT1` have after the if-statement completes execution? Considering what happened with the previous if-statement, what does that suggest about the way C handles typecasts in mixed-sign expressions?

---

2. [42 points] The file `q2.c` contains a short `main()` function and the following recursive function:

```
/** Computes a rather strange expression, recursively.
 *
 *  Pre:     x <= y
 *  Returns: something strange
 */
static int sumOfProducts(int x, int y) {

      // base cases
      if ( x == y )
         return x * y;
      if ( x + 1 == y )
         return (x + 1) * y;

      // recursive computation
      int next = x * y + sumOfProducts(x + (y-x)/2, y);

      // return when recursion backs out...
      return next;
}
```

Compile the given code for debugging, producing an executable named `q2`; it compiles without warnings.

a) Show what happens if you run the program with the parameters 3 and 19, in that order.

b) Now, start `gdb` on `q2` and set a breakpoint for the first **`return`** statement in the function `sumOfProducts()`, then run the program with the parameters 3 and 19 again. Copy enough of your `gdb` session to show exactly how you did this, including how you set the breakpoint, and what happened when you ran the program.

What does this tell you about what must have been true during that execution of the program?

c) Clear the breakpoint you set in the previous part. Set a new breakpoint for the second **`return`** statement in the function `sumOfProducts()`, and run the program again with the parameters 3 and 19. Copy enough of your `gdb` session to show exactly how you did this, including how you set the breakpoint, and what happened when you ran the program.

What were the values of `x` and `y` when your breakpoint was reached?

d) Clear the breakpoint you set in the previous part. Set a new breakpoint for the final **`return`** statement in the function `sumOfProducts()`, and run the program again with the parameters 3 and 19. Copy enough of your `gdb` session to show exactly how you did this, including how you set the breakpoint, and what happened when you ran the program.

What were the values of `x` and `y` when your breakpoint was reached?

e) **Do not clear the breakpoint you set in the previous part.** Now, enter the `continue` command in `gdb`. Copy enough of your `gdb` session to show exactly how you did this, and to show what happened.

In terms of the recursion, explain what `gdb` showed here.

f) **Do not clear the breakpoint you set part e).** Now, use the `continue` command enough times for the `q2` to terminate. This will result in pausing at the breakpoint a number of times, corresponding the recursion *backing out*. That will show information about the calls to `sumOfProducts()` that were made leading up to the state of the execution when you paused at the breakpoint in part d).

Altogether, from the beginning of the execution of `q2`, how many calls to `sumOfProducts()` were made, and what were the parameters passed to `sumOfProducts()` for each of those calls? You should consider the `gdb` output shown for some previous parts of this question, as well as the new `gdb` output.

---

3. **[28 points]** 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 retval = -1;
   int pass = 1;
   while ( pass < iterLimit ) {

      retval = q3(pass);           // convert
      pass++;
   }
   return 0;
}

int q3(int N) {

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

   dejavue = N * dejavue + dejavue % 67;  // perform strange computation

   return ( dejavue % 701 );  // return value between 0 and 700, inclusive
}
```

a) Use appropriate[2] `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 994. Show the relevant `gdb` output to support your answer.

b) Use appropriate `gdb` commands to analyze the program and determine what value the variable `pass` will have at the beginning of the first iteration on which `q3()` returns the value 888. Show the relevant `gdb` output to support your answer.

c) 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 the first time `q3()` returns the value 888. Show the relevant `gdb` output to support your answer.

Notes

[1] Adapt this command to compile the code that is given for questions 2 and 3.

[2] For this question, you'll have to explore using *conditional breakpoints* (see the GDB lecture notes for examples).

Change Log

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

Version	Posted	Pg	Change
5.00	March 2		Base document

Capturing gdb Sessions

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.

Having said all of that… I just copied from a Linux terminal window on my CentOS VM, and pasted into a text editor.