Exercises serial debugging

These exercises were originally developed for the Cyberinfrastructure Tutor, the web-based training site for High Performance Computing and Cyberinfrastructure topics:

`http://www.citutor.org`

which is hosted by the National Center for Supercomputing Applications (NCSA). The exercises and solutions are adapted to show the use of different debugging tools and compiler options besides using traditional debuggers.

Please note that using a different system, compiler or debugger — for that matter even a different version of the same compiler or debugger — could result in a somewhat different behavior of the debugger.

The provided Makefiles all use the gfortran compiler, but you can use the IBM compiler instead as follows:

```
xlf90_r -g -o program program.f90
```

Exercise 1: Checking variable values

The C code variablePrinting.c, is a very simple program that:

1. generates a 10-element array,
2. passes this array to a function that squares each element in the array and stores these values in a second 10-element array, and then
3. computes the difference between the element values in the two arrays.

We use this sample code to show an example debugging session using GDB to examine variable values. You will also learn how to set breakpoints, running and stepping through your code.

After compiling and running the sample code we obtain the following output:

```
$ gcc -o variablePrinting variablePrinting.c

$ ./variablePrinting

array1 = [  2  3  4  5  6  7  8  9  10  11  ]

The difference in the elements of array2 and array1 are: del = [ 0 0 0 0 0 0 0 0 0 ]
```

The result for the array *del* certainly does not look correct.

a. What are the expected values for array *del*?

(he fact that glibc detects a “double free or corruption” is also a hint that there is a coding error. We will come back to this at the end of the exercise.)

To identify this error we recompile our code using the debugging option '–g' and analyze it using GDB. In this example we illustrate how to debug a code by using a debugger to print out the values of selected variables during the execution of a program. You can get a short help on a command with 'help [command]'.

One possibility that could account for the values of all elements of *del* being zero is a coding error in our function squareArray(). If, for some reason, this function fails to square the elements of *array1* and, instead, is simply returning an array, *array2*, whose elements are identical to those of *array1* then del[k] = array2[k] – array1[k] = array1[k] – array1[k] = 0

for all *k*.

Therefore, we might first want to examine the values of the elements of the array *array2* to see if *{ a2k } = { a1k }*. We can do this in either of two ways. The first way is to set a breakpoint on the line of our code where we call the function squareArray() then print out the value of *array2[indx]* as we step through all *nelem* iterations of the for() loop within the body of this function.

```
int squareArray(const int nelem_in_array, int *array)
     {
       int indx;

       for (indx = 0; indx < nelem_in_array; indx++)
       {
         array[indx] *= array[indx];
       }
       return *array;
     }
```

The second way is to set a breakpoint immediately after the call to squareArray() and print out the elements of *array2* returned by the call to squareArray(). This example will illustrate both ways of examining these values *{ a2k }*.

After recompiling our code we run it using GDB.

```
$ gcc -g -o variablePrinting variablePrinting.c
$ gdb -tui ./variablePrinting
(gdb)
```

The tui option shows the source code in the upper half of the terminal, which is great to see where the execution of the program is, what will be the next command and to identify locations of breakpoints. You can see the source with the 'list' or 'l' command.

If we want to step into and through our function squareArray(), we need to set our initial breakpoint on line 37 and then rerun our code.

```
(gdb) b 37
Breakpoint 1 at 0x8048469: file variablePrinting.c, line 37.
(gdb) run
Starting program: variablePrinting

   array1 = [  2  3  4  5  6  7  8  9  10  11  12  13  ]


Breakpoint 1, main () at variablePrinting.c:37
37        squareArray(nelem, array2);
```

We can then step into this function by issuing the 'step' (or 's') command.

```
(gdb) s
squareArray (nelem_in_array=12, array=0x80498d8) at variablePrinting.c:68
68        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) s
70          array[indx] *= array[indx];
```

At this point we begin printing out the values of the elements of *array* on both the left- and right-hand side of the expression on line 70. We also track our progress thru the for() loop by printing out the value of the variable *indx* during each iteration of the loop. We can print the value of each of these variables by issuing the 'print' (or 'p') command followed by the name of the variable whose value you want to print; e.g.,

```
(gdb) p indx
$1 = 0
(gdb) p array[indx]
```

In this example, we are only printing out the values of two variables. However, as the number of variables becomes more than just a few, this method of printing a sequence of variable values will begin to require a lot of typing.

Fortunately, GDB provides two alternative methods for printing out a series of variable values without having to type multiple print commands each time you want to examine these values. One method involves creating a print macro issuing the GDB 'define' command. The second method involves issuing the GDB 'display' (or 'disp') command. The use of both of these methods to print out the values of *indx* and *array[indx]* during execution of the for() loop on line 33 is illustrated below.

First, we define a print macro named *var*.

```
(gdb) define var
```

After issuing this command GDB will then respond with the statement

```
Type commands for definition of "var".
End with a line saying just "end".
>
```

We then simply type a printf statement identifying the variables we want to include and the format in which we want each of these variables printed,

```
Type commands for definition of "var".
End with a line saying just "end".
>printf “indx = %d\n  array[indx] = %d\n”, indx, array[indx]
>end
(gdb)
```

Note that we notify GDB that our definition of *var* is complete by typing 'end'.

Now, each time we want to print out the values of *indx* and *array[index]* we simply type the name of our print macro, *var*.

```
(gdb) var
indx = 0
 array[indx] = 2
(gdb) s
68        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) var
indx = 0
 array[indx] = 4
(gdb) s
68          array[indx] *= array[indx];
(gdb) var
indx = 1
 array[indx] = 3
(gdb) s
68        for (indx = 0; indx < nelem_in_array; indx++)
(gdb) var
indx = 1
 array[indx] = 9
```

Although it is already obvious that our function squareArray() appears to be behaving properly; viz., the function is correctly computing the squares of the elements of array, we print out the values of our variables during the remaining iterations of this loop just to illustrate the use of the 'disp' command. For each variable we want to print out we type the command 'disp variable-name',

```
(gdb) disp indx
1: indx = 1
(gdb) disp array[indx]
2: array[indx] = 9
```

Once we have notified GDB of the variables we want displayed, GDB will automatically print out the value of these variables each time the program pauses; e.g., each time we issue the step (s) command to step through our for() loop as illustrated below.

```
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 4
1: indx = 2
(gdb) s
67          array[indx] *= array[indx];
2: array[indx] = 16
1: indx = 2
(gdb) s
65        for (indx = 0; indx < nelem_in_array; indx++)
2: array[indx] = 5
1: indx = 3
(gdb)
```

Clearly, using the 'disp' command for printing multiple variables requires even less typing than using a 'print' macro. However, since each of the variables are automatically printed every time the program pauses, it is most useful only when examining variables whose values are changing frequently; e.g., during the execution of a for() loop. Otherwise, we would probably not want to see a set of constant values being printed out over and over throughout an entire debugging session. Fortunately, there is an easy way to tell GDB to stop displaying a given variable. We simply invoke the command 'undisp' followed by the number *n* that is printed to the left of the variable name,

n: variable-name = value

For example, we can cancel the printing of the variables *indx* and *array[indx]* by issuing the commands

```
(gdb) undisp 1
(gdb) undisp 2
(gdb)
```

Examining the values of *array[indx]* we see that each element of *array2* returned by squareArray() is equal to the square of the corresponding element in *array1*. Consequently, we know that our coding error is not located anywhere within the function squareArray(). Therefore, we need to continue debugging our program until we locate the coding error in our program.

Use the 'finish' command to continue until the end of the squareArray() function.

Complete the debugging of this code and identify the coding error in the program.

(b. How can you print the whole array array1 in one go?)

(c. The fact that glibc detects a “double free or corruption” is also a hint that there is a coding error. Does glibc tell you more about the location of the error? If this happens in a job, what tool could you use to find the associated line in the source code?)

Exercise 2: arrays

The default range of indices differs among the major programming languages. For example, in

Fortran 77 the default index range is *i=1,...,N* (where *N* is the array size); in C/C++ the default range is *i=0,...,N-1*; and in Fortran 90 the allowed index range can be *any* sequence of integers with *any* spacing. So if you routinely use several programming languages, it is easy to confuse the indexing rules.

After compiling and executing the code, we get the following result:

```
debug$ gcc -o array array.c
debug$ ./array
oddsum=5
evensum=224683
debug$
```

As you can see, the code does not generate compiler or runtime errors. However, something is wrong. By looking at the body of the first loop it looks like the array elements should be relatively small integers. So the value of *oddsum* is probably alright. But why is the value of *evensum* so large? The value of *evensum* is computed in our summation loop, so we suspect that the error occurs there. Therefore, we will concentrate our debugging effort on that section of code.

There are five basic steps we will take to identify the error along with a final step of exiting the debugger:

1. Recompile using the '-g' option
2. Run the debugger (you can use gdb or TotalView, whichever you prefer)
3. Identify the breakpoint
4. Execute the code up to the breakpoint
5. Step through the summation loop

The first two steps shouldn't be a problem anymore. Let's put a breakpoint at the beginning of the summation loop.

a. What's the command to do that? How do we run the program up to the breakpoint?

Before analyzing the summation loop, it is good idea to check out some of the array elements to see if they have the correct values.

b. Use the 'print' debugger command to show the values of the elements `tock[2]` and `tock[7]`. Are the values what you expected?

c. How do we display the value of *evensum* to make sure that it did get initialized to zero and how it gets updated?

Step through the summation loop until you see the problematic values.

d. What is the cause of the random `evensum` and how can you solve this?

e. How can you print the whole array *tock* in one go?

Exercise 3: learning valgrind

a. use valgrind to run the program variablePrinting from the first exercise. Valgrind finds two problems in our code, which ones? On which lines did these problems occur? Does this match the errors that you found in the first exercise?

b. use valgrind to run the program array from the second exercise. Does it detect the use of uninitialised values in your arrays? Although we only had one undefined value, valgrind seems to find several more. Why is that? What is the origin of the uninitialised values? See also http://valgrind.org/docs/manual/mc-manual.html#mc-manual.uninitvals

Exercise 4: Dynamic summation

The program readsum.c has an indexing error in it. The code reads in data from a file called input.txt and puts it in the array *tock*. Then the sum of all the elements is calculated and printed out.

The code compiles, run, and even produces the following output:

```
$ gcc -o readsum readsum.c
$ ./readsum < input.txt
sum is 1090515646
$
```

As you can see from the output, the value of the sum is way too large given the numbers input. We leave it up to you to figure out why.

Hint: to run an application in gdb that reads the file input.txt from standard input, use

```
$ gdb ./readsum
(gdb) start < input.txt
```

Exercise 5: Fortran I/O

For this lesson we use an the example program sections.f90 written in Fortran 90 that

1. reads in a set of data stored in ASCII format
2. does some simple data manipulation
3. writes the results as a binary file
4. reads the binary file back in before finally writing some of the fields to a formatted text file

As you will see, this program is very poorly written and full of errors.

When our sample code is compiled using the XLF compiler and run, it gives the following error:

```
$ sections
1525-001 The READ statement
on the file sectionsCT.txt cannot be completed because the end
of the file was reached.  The program will stop.
```

There are two possible and common reasons why the XLF compiler gives this error message:

1. the program is trying to read more data than there is in the file
2. the file "sectionsCT.txt" does not even exist.

It's a good practice to open files read-only when you don't intend to write to it. This prevents confusing error messages or accidentally overwritten files. So input-files should be opened only when they exist, and output files should be opened only if they don't already exist.

a. What is the optional argument to only open existing files? What happens when a file is opened that does not exist?

Edit the source code so it only opens only existing files and run 'make clean; make'. Now, when we run the program the runtime environments fails with the helpful message that the file does not exist.

Correct the file name in the source code, recompile and rerun the program. The fortran runtime environment runs into more errors when reading the input file.

b. Compare the input that is read in the program to what you expect it should be reading (either through print statements or a debugger). When printing strings, it is often difficult to see if there are extra spaces at the beginning or end, how can you make these visible?

c. Edit the format to correctly parse the input-file.

Rerun the program and check that the output in the output file is as expected.

d. Is it correct? What went wrong?

Exercise 6: argument and type checking

When a subroutine is called from within a program, the program call's argument list must match that of the subroutine it is calling. Since typically there are multiple items in a subroutine's argument list it can be easy to inadvertently create a mismatch between the subroutine's dummy argument list and the actual argument list passed by the corresponding program call. Some of the more frequent types of argument mismatches are:

- The number of terms in the argument list do not match
- The data types of the terms in the argument list do not match
- Data pass-by-value mismatch (When a C program calls a Fortran subprogram, all variables (including scalars) must be passed by reference instead of by value.)

Compilers react quite differently to these mismatches. Some may issue a warning while others may not. For C programs that use prototyping and Fortran 90 programs that declare subroutines via the f90 module interface, the compiler can readily perform an argument list compatibility check to see if there are any mismatches.

If you prefer Fortran, then rewrite the program miss1.f90 to a Fortran-90 program using modules.

If you prefer C, then rewrite miss1.c to use prototyping.

Check that the compiler actually detects the incorrect use of arguments.

Exercise 7: C++ pointer arithmetic

Since the name of an array is simply an address (specifically, the address of the first element in the array), then we can always point to the beginning of an array of elements using a pointer. The statement

ptr = arr;

initializes a pointer, *ptr*, to point to *&arr[0]*, where *arr* is the name of some array.

If we want to write a function that returns the address of the beginning of the array *arr*, one way to do this is to have the function return the name of the array, *arr*. An alternative way would be to initialize a pointer, *ptr*, using the statement shown above and have the function return the pointer, *ptr*.

The C++ program pointers.cc uses this concept to copy the elements from one array to another array. Run this program to see if it completes successfully and, if not, debug the program in the debugger of your choice, identify the bug(s) in the code, and see if you can correct the error(s) so that the program runs successfully.

Note that in the code pointers.cc we are using the ANSI-C++ standard header files.

Compile and run the program. Choose your favorite tool to debug this problem:

- addr2line
- valgrind
- ElectricFence
- gdb
- totalview

Exercise 8: real numbers

The Fortran code trapezoid.f90 integrates the function cos(x) over the interval zero to pi using the trapezoidal rule. The result should be identically equal to zero. For demonstration purposes, only 10 intervals (11 endpoints) are used.

The sample code was compiled and run, and gave the following output:

```
  i        x       f(x)    int(f(x))
  1      0.00      1.00      0.00
  2      0.00      1.00      0.00
  3      0.00      1.00      0.00
  4      0.00      1.00      0.00
  5      0.00      1.00      0.00
  6      0.00      1.00      0.00
  7      0.00      1.00      0.00
  8      0.00      1.00      0.00
  9      0.00      1.00      0.00
 10      0.00      1.00      0.00
 11      3.14     -1.00      0.00
```

The four columns contain the endpoint number, the independent variable *x*, the dependent variable *cos(x)*, and the integral of the function from the left endpoint to the current point. The good news is that the final value of the integral is exactly zero, as expected. The bad news is that most of the other values in the output file are clearly incorrect.

a. Use your favourite debugger to find why the value of x does not advance through the loop.

b. Could the GNU compiler be of help to locate these issues for you?

Exercise 9: dynamic memory

Static memory allocation has been the traditional method for assigning memory to an array for quite some time. In this method, memory is allocated at the beginning of the program and does not change for the duration of the program. One problem with this method is that the size of the array may not be known at the initial declaration. Sometimes the array size will be input from a file or user prompt or perhaps be calculated in the program. In these cases, memory cannot be statically allocated to the array.

Dynamic memory allocation is the ability to assign memory to an array at any point in a program. In dynamic memory allocation, the memory size can be determined from sources such as a file, user prompt, or calculation and memory of the appropriate amount is connected to the array name at the point in the code when it is needed. Unlike static memory allocation, this method does not tie up all the memory allocated to it throughout the duration of the program but rather just from the point at which it is allocated. In addition, the size declared is the exact size needed for the array and the memory can be released when the array is no longer needed.

Several high-level languages provide the ability to dynamically allocate memory. These languages typically provide this capability through built in routines that can be called when memory needs to be allocated. The languages most commonly used for dynamic memory allocation are C, C++, and Fortran 90.

As with any programming technique, when dynamic memory allocation is performed incorrectly it results in a variety of errors. Also, how the error manifests itself depends on which compiler, loader, and machine is used. In one situation, the compiler could catch the programming error and in another an executable could be made. In the latter case, when the buggy code is executed either a run-time error occurs or incorrect results are produced.

Two of the most common programming errors in dynamic memory allocation are trying try to allocate too much memory and not allocating the memory at all. Also common are allocating a zero size, the incorrect size, and for multi-dimensional arrays not allocating enough memory for all the dimensions. This is by no means an exhaustive list of the errors that can occur but it should give you an idea of the most common ones.

Try the program dynamic.f90 using both the IBM and GNU compilers.

While the executable compiled with the GNU compiler gives a segmentation fault, the IBM compiler gives you the wrong result.

a) Can you find the problem in the executable compiled with the IBM compiler with a memory debugging tool like valgrind or Electric Fence? Why do you think is that?

Of course, the incorrect result is quite obvious here, but you might not notice it directly in a large simulation.

b) What optional flag could you use with the IBM compiler to check for incorrect use of arrays?

c) With the knowledge in mind that different compilers could behave quite differently when compiling and running the application, how could you improve the robustness of your application?