Virtual Summer School: debugging

Erik Spence

SciNet HPC Consortium

30 August 2021

SciNet compute • calcul CANADA

Material for this class

The slides and code for this class can be found here:

`https://scinet.courses/584`

All the material for the 2021 Virtual Summer School can be found here:

`https://scinet.courses/573`

Make sure that you confirm your attentance before the end of class.

Details about this course

To get credit for this course:

- You need to attend 2 out of 3 sessions.
  - Each session is about 1.5 hours,
  - Sessions are 12:30 - 2:00pm, Monday, Wednesday, Friday, August 30 - September 3.
  - To demonstrate attendance, you must take the attendance test each class.
- You must submit the assignment.

This course is based on material developed by Bruno Mundim, and other SciNet analysts.

Ask questions!

SciNet compute • calcul CANADA

Today's class

The purpose of this class is to introduce you to the basics of debugging high-performance code. We will cover the following topics:

- Debugging Basics
- Debugging with the command line: GDB
- Memory debugging with the command line: Valgrind

Note that this class will be exclusively about compiled languages (C, C++, Fortran). We will not be looking at debugging interpreted languages (Python, R) today.

Debugging basics

Help, my program doesn't work!

a miracle occurs

My program works brilliantly!

```
ejspence@mycomp ~> gcc -O3 answer.c
ejspence@mycomp ~> ./a.out
Segmentation fault
```

```
ejspence@mycomp ~> gcc -O3 answer.c
ejspence@mycomp ~> ./a.out
42
```

Unfortunately, miracles are not yet supported by SciNet.

Debugging: the methodical process of finding and fixing flaws in software.

Common types of errors

There are two broad modes of failure.

- Compile-time errors (errors which occur during compilation):
  - Syntax errors: easy to fix,
  - Library issues (linking, missing libraries, missing objects),
  - Compiler warnings (always switch this on, and fix or understand them!),
- Runtime errors (errors which occur when the code is run):
  - floating point exceptions,
  - segmentation faults,
  - code is aborted,
  - incorrect output (NaNs),

Debugging generally focusses on runtime errors. Just because your code compiles does not mean it is correct!

How to avoid debugging

Of course, you're better off not needing to debug at all. Some tips:

- Write better code.
  - Write simple, clear, straightforward code.
  - Use sensible variable and function names.
  - Comment your code!
  - Write modular code (no global variables or 10,000-line functions).
  - Write testing routines for your functions.
  - Avoid 'cute' tricks (no obfuscated C code winners).
- Use version control so you can 'roll back' your code when things go wrong.
- Be systematic! Take note of your assumptions.
- Don't write code, use existing libraries whenever possible.

Bugs will still creep in, but using good coding best-practices will keep the bugs to a minimum, and make them easier to find!

Debugging workflow

Ok, so you've written and compiled your code, but it crashes when it runs. What to do?

- As soon as you are convinced there is a real problem, create the simplest example in which the crash consistently occurs.
- Be methodical: model, hypothesis, experiment, conclusion.
- Try a smaller problem size, turning off compilation flags, etc., until you have a simple, fast repeatable example of the bug.
- Try to narrow it down to a particular module/function/class.
  For fortran, switch on bounds checking (**-fbounds-check**.)

Now that you've narrowed down the problem you're ready to start debugging.

SciNet compute • calcul CANADA

Ways to debug

There are several approaches to debugging.

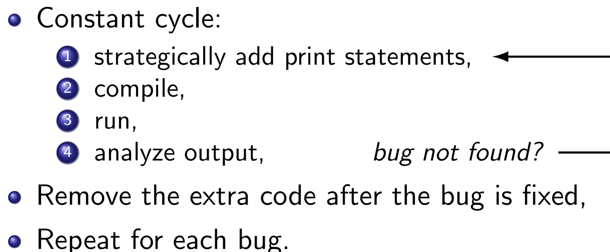
- Preemptive:
  - Turn on compiler warnings (for gcc, use the **-Wall** flag). These are not just there to annoy you, they give useful information. Fix the causes of the warnings, or at least understand them!
  - Check your assumptions (e.g. use **assert** command).
- If your code is crashing, inspect the exit code and read the error messages!
- Use a debugger. This a program specially designed for finding bugs in programs.
- Add print statements. ←Bad!

Adding print statements is not the correct way to debug a code.

What's wrong with print statements?

Strategy:

- Constant cycle:



1. strategically add print statements,
2. compile,
3. run,
4. analyze output, *bug not found?*

- Remove the extra code after the bug is fixed,
- Repeat for each bug.

Problems with this approach:

- Time consuming,
- Error prone,
- Changes memory, timing...

At the best of times this is inefficient. There's a better way!

Symbolic debuggers

Debuggers (also called symbolic debuggers), are programs which have been specifically designed to find bugs in programs. These programs are used for

- code crash inspection,
- examination of the function call stack,
- stepping through the code,
- variable checking and setting.

Some debuggers have a graphical user interface. Should I use a graphical debugger or not? That depends on where you're working:

- On your local work station: graphical is convenient,
- Remotely, over an internet (SciNet): can be slow, unless the debugger supports a client-server mode.

In any case, graphical and text-based debuggers use the same concepts.

Symbolic debuggers, continued

So, how do we go about using a debugger?

- The first step is to prepare the executable.
  - Recompile the code with the required compilation flags. This adds symbols into the executable which the debugger uses to understand what is going on. The most commonly used debugging flag is **-g**.
  - Optional: switch off code optimization **-O0**.
- Launch the code using the debugger.

The specific debugger compilation flags depend upon the compiler being used.

```
ejspence@mycomp ~> gcc/g++/gfortran -g [-gstabs]
ejspence@mycomp ~> icc/icpc/ifort -g [-debug parallel]
ejspence@mycomp ~> nvcc -g -G
```

The next step is to launch the code using the debugger.

SciNet compute • calcul CANADA

What is GDB?

We will use GDB, a commonly used debugger (GNU DeBugger).

- Free, Open Source, GNU license, symbolic debugger.
- Available on most Unix-like systems. Available on SciNet's Teach cluster and Mist as a module, built into Niagara.
- Works with many languages (C, C++, Fortran, Ada, Go, Julia, Python).
- Has been around for a long time, but is still being actively developed.
- Text based, which means it's easy to use in a terminal.
- But it also has a '-tui' (Text User Interface) option.

GDB is a excellent tool for debugging. We will run some examples on SciNet's Teach cluster, but you can also install GDB on your local machine and use it there.

SciNet compute • calcul CANADA

Installing GDB

If you want to install GDB on your local machine, you have a few options.

- The official download website is found here: `https://www.gnu.org/software/gdb/download`. However, this only leads to the source code.
- On a Mac, use brew or fink.
- On a Windows 10 machine,
  - use the Windows subsystem for Linux (Ubuntu). Use apt-get to install.
  - some other method? Good luck!
- On Linux, use apt-get, yum or your favourite package manager to install GDB.

Download the code from the class web site so you can run the examples on your local machine. Note that we will also use the "Valgrind" code, so it's worth your time to install both.

SciNet compute • calcul CANADA

Getting set up on SciNet

Be sure to use your username, not ejspence, when you attempt to log into SciNet.

```
ejspence@mycomp ~> ssh -X ejspence@teach.scinet.utoronto.ca
ejspence@teach01 ~>
ejspence@teach01 ~> cd $SCRATCH
ejspence@teach01 scinet/ejspence>
ejspence@teach01 scinet/ejspence> cp -r /scinet/course/ss2021/8_debug .
ejspence@teach01 scinet/ejspence>
ejspence@teach01 scinet/ejspence> cd 8_debug
ejspence@teach01 ejspence/8_debug>
```

These steps are only necessary if you want to follow along with the examples on the teach cluster. If you're following along on your local machine you should download today's code from the class web site.

Do NOT try to copy-and-paste code out of PDFs!

SciNet compute • calcul CANADA

GDB example

Let's examine the features of GDB by doing an example.

- The example code reads integers from the command line and sums them.
- The example is written in C.

When doing the below commands, make sure you've logged into the teach cluster, and have copied the class code into your SCRATCH directory (see the previous slide).

```
ejspence@teach01 ejspence/8_debug>
ejspence@teach01 ejspence/8_debug> source setup
ejspence@teach01 ejspence/8_debug>
ejspence@teach01 ejspence/8_debug> cd bugexample/c
ejspence@teach01 bugexample/c>
ejspence@teach01 bugexample/c> make bugexample
ejspence@teach01 bugexample/c>
```

SciNet compute • calcul CANADA

GDB example, continued

What happened?

- The program crashed with a "Segmentation fault".
- This usually means that it attempted to access an illegal memory location.
- The "unlimit -c 1024" controls the maximum core file size that will be produced.
- The "core file" contains the in-memory state of the program at the time that it crashed (a "core dump").
- The core file is named "core" or "core.XXXX".

```
ejspence@teach01 bugexample/c>
ejspence@teach01 bugexample/c> ./bugexample
Give some integers as command-line arguments
ejspence@teach01 bugexample/c>
ejspence@teach01 bugexample/c> ./bugexample 1 3 5
Segmentation fault
ejspence@teach01 bugexample/c>
ejspence@teach01 bugexample/c> ulimit -c 1024
ejspence@teach01 bugexample/c>
ejspence@teach01 bugexample/c> ./bugexample 1 3 5
Segmentation fault (core dumped)
ejspence@teach01 ~>
```

GDB can use the core file to diagnose the program.

SciNet compute • calcul CANADA

GDB example, inspecting the core file

GDB will inspect the core file to try to figure out what went wrong.

- Core = file containing state of program after a crash.
  - GDB reads the file by using the command **gdb <executable> <corefile>**
  - It will show you where the program crashed.
- No core file?
  - You can start GDB using **gdb <executable>**
  - Type **run** (with any needed arguments) to start the program.
  - GDB will show you where the program crashed if it does.
- Related GDB commands:
  - run: run the executable from the start.
  - list: list code lines (where current execution is, or range).

Once GDB has been started it will create an interactive prompt which allows further commands to be entered.

SciNet compute • calcul CANADA

GDB example, inspecting the core file, continued

The code failed at line 30 of the intlisttools.c file.

When running GDB, the program's "location" is the beginning of the next line of code to be run by the program. GDB's current location is line 30.

If you ever want to get out of the GDB prompt, type "quit", or Ctrl-D.

```
ejspence@teach01 bugexample/c>
ejspence@teach01 bugexample/c> gdb ./bugexample core.2387
GNU gdb (GDB) 7.6
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from bugexample/bugexample...done.
[New LWP 3817]
warning: Can't read pathname for load map: Input/output
error.
Core was generated by `./bugexample 1 3 5'.
Program terminated with signal 11, Segmentation fault.
#0 0x4007d5 in sum_integers (n=3, a=0x4)
at intlisttools.c:30
30 s += a[i];
(gdb)
```

Symbolic debuggers, an aside

Those of you who have never used a symbolic debugger before, take note:

- Notice that we are now *inside* the program *while it is running*!
- True, the program has been paused for the moment, but we can restart it when we want.
- Since we are inside the program, we can poke around, examine variables, check assumptions, etc.
- We can directly examine what the program thinks is going on.
- No print statements, or recompiling, are needed!

Using a debugger is the correct way to debug a compiled code.

SciNet compute • calcul CANADA

GDB example, inspecting the core file, continued more

We can use GDB's "list" command to print out the code around its current location.

The line numbers of the file are listed on the left.

Note that if you hit 'Enter' at the GDB prompt without a command GDB will repeat the last command entered.

```
(gdb)
```

```
(gdb) list
25      /* Compute the sum of the array of integers */
26      int sum_integers(int n, int* a)
27      {
28         int i, s;
29         for (i=0; i<n; i++)
30            s += a[i];
31         return s;
32      }
```

```
(gdb)
```

If list has already been run once, the 'list' command will continue to list code starting at the end of the last list output.

SciNet compute • calcul CANADA

GDB example, the function call stack

When running the program from within GDB, you can interupt the program mid-execution:

- Press Crtl-C while program is running in GDB,
- GDB will show you where the program was.

You can also get access to the "stack trace":

- From what functions was this line reached?
- What were the arguments of those function calls?

More GDB commands:

frame	f	print the current line
backtrace	ba	print the function call stack
continue	c	continue execution
down	do	go to called function
up	up	go to calling function

SciNet compute • calcul CANADA

GDB example, the function call stack, continued

```
(gdb)
```

```
(gdb) frame
#0 0x00000000004007ac in sum_integers (n=3, a=0x4) at intlisttools.c:30
30 s += a[i];
```

```
(gdb)
```

```
(gdb) backtrace
#0 0x4007d5 in sum_integers (n=3,a=0x4) at intlisttools.c:30
#1 0x40082a in process (argc=4,argv=0x7fff0b89ce58) at process.c:11
#2 0x4006d3 in main (argc=4,argv=0x7fff0b89ce58) at bugexample.c:12
```

```
(gdb)
```

You can use the "frame" command to remind yourself of where you are in the code.

SciNet compute • calcul CANADA

GDB example, variables

GDB allows you to inspect the program's variables.

- Can print the value of a variable,
- Can keep track of variable (print at prompt),
- Can stop the program when variable changes,
- Can change a variable ("what if ...").

More GDB commands:

print variable	p	print the variable's value
display variable	disp	print the variable's value at every prompt
set variable	set var	change the variable's value
watch	wa	stop if the variable's value changes

GDB example, variables, continued

```
(gdb)
```

```
(gdb) list 25, 32
25      /* Compute the sum of the array of integers */
26      int sum_integers(int n, int* a)
27      {
28         int i, s;
29         for (i=0; i<n; i++)
30            s += a[i];
31         return s;
32      }
```

```
(gdb) print i
$1 = 0
```

```
(gdb) print a[0]
Cannot access memory at address 0x4
```

```
(gdb) print a
$2 = (int *) 0x4
```

```
(gdb)
```

GDB example, variables, continued more

```
(gdb)
```

```
(gdb) up
#1 0x000000000040082a in process (argc=4, argv=0x7fff0b89ce58) at process.c:11
11 int s = sum_integers(n, arg);
```

```
(gdb) print arg
$3 = (int *) 0x4
```

```
(gdb) list
7       void process(int argc, char** argv)
8       {
9          int* arg = read_integer_arguments(argc, argv);
10         int n = argc-1;
11         int s = sum_integers(n, arg);
12         print_integers(n, arg);
13         printf("Sum of integers is: %d\n", s);
14      }
```

SciNet compute • calcul CANADA

GDB example, automatic interruption

GDB allows you to insert "breakpoints" into the code:

- **break [file:]<line>|<function>**
- each breakpoint gets a number
- when run, GDB automatically stops the program there
- you can also add conditions, temporarily remote breaks, etc.

More GDB commands:

delete	d	unset a breakpoint
condition	cond	break if a condition is met
disable	dis	disable a breakpoint
enable	en	enable a breakpoint
info breakpoints	inf b	list all breakpoints
tbreak	tb	temporary breakpoint

SciNet compute • calcul CANADA

GDB example, automatic interruption, continued

Here we set a breakpoint at the read_integer_arguments function.

We then rerun the program with arguments "1 3 5". The program then run until it either crashes, or hits the breakpoint.

```
(gdb) list 7, 14
7       void process(int argc, char** argv)
8       {
9           int* arg = read_integer_arguments(argc, argv);
10          int n = argc-1;
11          int s = sum_integers(n, arg);
12          print_integers(n, arg);
13          printf("Sum of integers is: %d\n", s);
14      }
```

```
(gdb) break read_integer_arguments
Breakpoint 1 at 0x4006ec:  file intlisttools.c, line 8.
```

```
(gdb) run 1 3 5
Starting program:  bugexample/c/bugexample 1 3 5
Breakpoint 1, read_integer_arguments (n=4, a=0x7fffffffc9b8)
at intlisttools.c:8
8 int* result = malloc(sizeof(int)*(n-1));
```

```
(gdb)
```

GDB example, stepping through the code

You can also use GDB to step through the code, stopping after every step:

- You can step line-by-line.
- You can choose to step into or over functions.
- You can show surrounding lines or use the **-tui** flag.

More GDB commands:

list	l	list part of code
next	n	continue until next line
step	s	step into function
finish	fin	continue until the function's end
until	unt	continue until line/function

SciNet compute • calcul CANADA

GDB example, stepping through the code, continued

Here we list lines 6-14 of the current file.

Recall that the "display" command displays the value of a variable after every command.

The "next" command continues executing the code until the next line. Notice how the new value of the "result" variable is displayed.

```
(gdb)
(gdb) list 6, 14
6       int* read_integer_arguments(int n, char** a)
7       {
8          int* result = malloc(sizeof(int)*(n-1));
9          int i;
10         /* convert every argument, but skip '0', because it
11            is just the executable name */
12         for (i=1;i<n;i++)
13            result[i] = atoi(a[i]);
14      }
(gdb) display result
1:  result = (int *) 0x0
(gdb) next
12 for (i=1;i<n;i++)
1:  result = (int *) 0x601010
(gdb)
```

GDB example, continued more

Recall that "until" runs the program until a specific line or function is encountered. In this case we run until line 14.

We then run "finish" to run until the end of the function.

```
(gdb)
```

```
(gdb) until 14
read_integer_arguments (n=4,a=0x7fffffffc9b8) at intlisttools.c:14
14 }
1:  result = (int *) 0x601010
```

```
(gdb) finish
Run till exit from #0 read_integer_arguments (n=4,
a=0x7fffffffc9b8) at intlisttools.c:14
0x000000000040080c in process (argc=4, argv=0x7fffffffc9b8)
at process.c:9
9 int* arg = read_integer_arguments(argc, argv);
Value returned is $4 = (int *) 0x4
```

```
(gdb)
```

Notice that the result variable equal to **0x601010** while the value returned is **0x4**. Clearly something is not correct.

GDB example, continued even more

Why is the result variable equal to **0x601010** while the value returned is **0x4**?

```
(gdb)
-----------------------------------------------------------
(gdb) list read_integer_arguments,+7
7       {
8           int* result = malloc(sizeof(int)*(n-1));
9           int i;
10          /* convert every argument, but skip '0', because it
11             is just the executable name */
12          for (i=1;i<n;i++)
13             result[i] = atoi(a[i]);
14      }
-----------------------------------------------------------
(gdb)
```

Aargh! Forgot the return statement!

Feeling like an idiot is a common side-effect of debugging.

GDB example, summary

This example demonstrated the basics of what you can do with GDB.

- We used the core file to get GDB started.
- We stepped into the code.
- We printed out the values of variables.
- We printed out the stack trace.
- We printed out blocks of code.
- We popped out of a function.
- We created a breakpoint.
- We stepped through the code.

Use a debugger!

SciNet compute • calcul CANADA

GDB command summary

help	**h**	print description of command
run	**r**	run from the start (+args)
backtrace/where	**ba**	function call stack
list	**l**	list code lines
break	**b**	set breakpoint
delete	**d**	delete breakpoint
continue	**c**	continue
step	**s**	step into function
next	**n**	continue until next line
print	**p**	print variable
finish	**fin**	continue until function end
set variable	**set var**	change variable
down	**do**	go to called function
until	**unt**	continue until line/function
up	**up**	go to caller
watch	**wa**	stop if variable changes
quit	**q**	quit gdb

SciNet compute • calcul CANADA

Memory Checking: Valgrind

Sometimes you suspect (or know!) that your code has issues with memory.

- Memory errors do not always give Segmentation Faults ("segfaults").
- You commonly have to go *way* out of bounds to get a segfault.
- If you accidentally write into another variable it can very hard to find the problem.
- Valgrind is a debugging program which intercepts each memory call and checks them for correctness.
- Valgrind is free, open-source.
- It finds illegal memory accesses, uninitialized values, memory leaks, and other memory-use problems.
- Warning: the output of Valgrind can be quite verbose, an, if you use external libraries you can sometimes get false positives.

Valgrind is an excellent debugger if you have memory problems.

SciNet compute • calcul CANADA

Valgrind example

```
ejspence@teach01 bugexample/c> valgrind ./bugexample 1 3 5
==909== Memcheck, a memory error detector
==909== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==909== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==909== Command:  ./bugexample 1 3 5
==909==
==909== Invalid write of size 4
==909== at 0x400741:  read_integer_arguments (intlisttools.c:13)
==909== by 0x40080B: process (process.c:9)
==909== by 0x4006D2:  main (bugexample.c:12)
==909== Address 0x51c304c is 0 bytes after a block of size 12 alloc'd
==909== at 0x4C2636D: malloc (vg_replace_malloc.c:291)
==909== by 0x4006FF: read_integer_arguments (intlisttools.c:8)
==909== by 0x40080B: process (process.c:9)
==909== by 0x4006D2:  main (bugexample.c:12)
==909==
==909== Invalid read of size 4
...
```

Valgrind example (continued)

```
==909== HEAP SUMMARY:
==909== in use at exit:  12 bytes in 1 blocks
==909== total heap usage:  1 allocs, 0 frees, 12 bytes allocated
==909==
==909== LEAK SUMMARY:
==909== definitely lost:  12 bytes in 1 blocks
==909== indirectly lost:  0 bytes in 0 blocks
==909== possibly lost:  0 bytes in 0 blocks
==909== still reachable:  0 bytes in 0 blocks
==909== suppressed:  0 bytes in 0 blocks
==909== Rerun with --leak-check=full to see details of leaked memory
==909==
==909== For counts of detected and suppressed errors, rerun with:  -v
==909== ERROR SUMMARY: 2 errors from 2 contexts (suppressed:  6 from 6)
Segmentation fault
ejspence@teach01 bugexample/c>
```

Valgrind recommendations

Note that Valgrind gives a report on the code, rather than allowing you to actively explore it, like GDB. Some tips if you find yourself using Valgrind:

- Using Valgrind on mature codes often shows lots of errors. Now, some may not be an issue (e.g. dead code or false positives from libraries), but it's hard to know.
- So: start using Valgrind early in development.
- Program modularly, and create small unit tests, on which you can comfortably use Valgrind.
- Apart from this basic Valgrind usage, there are other tools availble with Valgrind to deal cache performance, to get more detailed memory leak information, to detect race conditions, etc.

We won't be spending more time on Valgrind today, but be aware it is an excellent memory-debugging tool.

SciNet compute • calcul CANADA

Useful references

There are many many useful tutorials on using GDB and Valgrind; a quick internet search will reveal many. Here are some basic references.

- N Matloff and PJ Salzman

  The Art of Debugging with GDB, DDD and Eclipse
- *GDB:* `https://www.gnu.org/software/gdb`
- *Valgrind:* `https://www.valgrind.org`
- *SciNet Wiki:* `https://docs.scinet.utoronto.ca/index.php/Performance_And_Debugging_Tools:_Niagara`

SciNet compute • calcul CANADA