Scientific Computing (PHYS 2109/Ast 3100 H)
I. Scientific Software Development

SciNet HPC Consortium
University of Toronto

Winter 2014

Lecture 6

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

Debugging basics

Debugging basics

Debugging basics

Help, my program doesn't work!

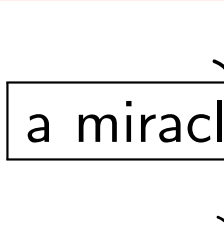
Debugging basics

Help, my program doesn't work!

```
$ gcc -O3 answer.c
$ ./a.out
Segmentation fault
```

Debugging basics

Help, my program doesn't work!



a miracle occurs

```
$ gcc -O3 answer.c
$ ./a.out
Segmentation fault
```

Debugging basics

Help, my program doesn't work!

a miracle occurs

My program works brilliantly!

```
$ gcc -O3 answer.c
$ ./a.out
Segmentation fault
```

Debugging basics

Help, my program doesn't work!

```
$ gcc -O3 answer.c
$ ./a.out
Segmentation fault
```

↓

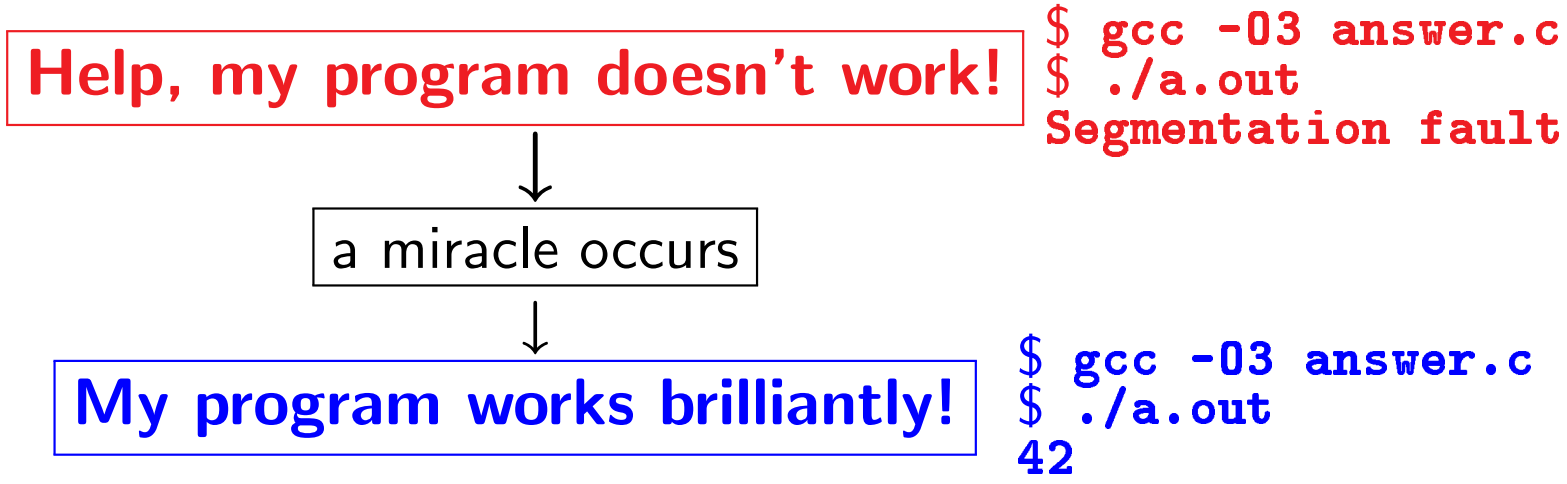
a miracle occurs

↓

My program works brilliantly!

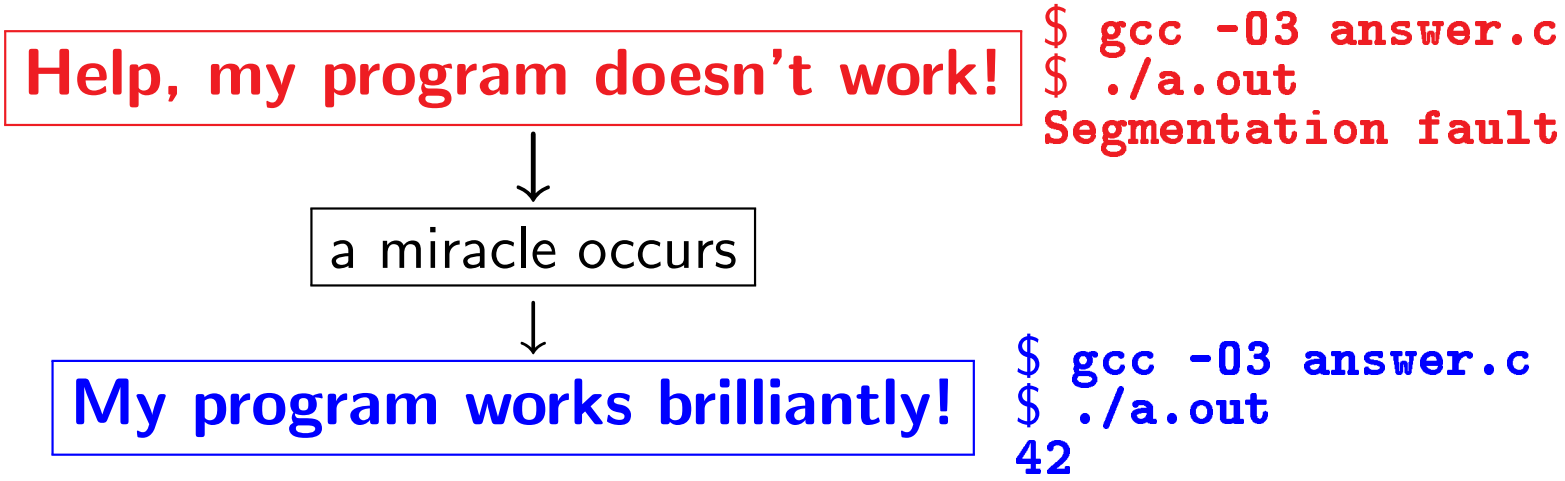
```
$ gcc -O3 answer.c
$ ./a.out
42
```

Debugging basics



- Unfortunately, "miracles" are not typically deterministic.

Debugging basics



- Unfortunately, "miracles" are not typically deterministic.

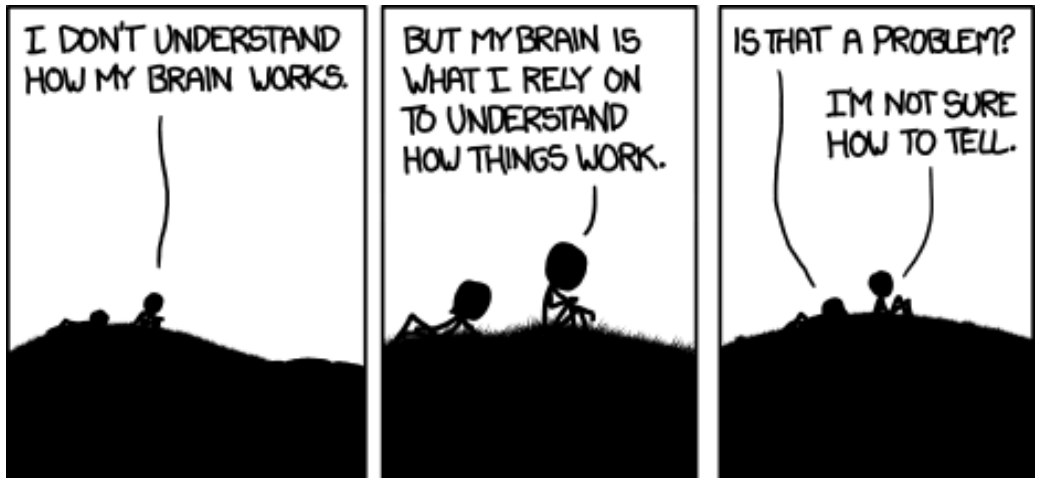
Debugging:

Methodical process of finding and fixing flaws in software

What is going on?

- All programs execute correclty.
- We just told it to do the wrong thing.
- Debugging is the art of reconciling your mental model of what the code is doing with what you actually told it to do.
- *Debugger:* program to help detect errors in other programs.
- **You are the real debugger.**

Debugging



http://imgs.xkcd.com/comics/debugger.png

Tips to avoid debugging

- Write better code.
  - simple, clear, straightfoward code.
  - modularity (avoid global variables and 10,000 line functions).
  - avoid "cute tricks", (no obfuscated C code winners).
- Don't write code, use existing libraries
- Write tests (simple) for each part

Debugging Workflow

- As soon as you are convinced there is a real problem, create the simplest situation in which it repeatedly occurs.
- This is science: model, hypothesis, experiment, conclusion.
- Try a smaller problem size, turning off different physical effects with options, etc, until you have a simple, fast, repeatable example.
- Try to narrow it down to a particular module/function/class.
- Integrated calculation: Write out intermediate results, inspect them.

Common symptoms

Errors at compile time

Common symptoms

Errors at compile time

- Syntax errors: easy to fix
- Library issues
- Cross-compiling
- Compiler warnings

Common symptoms

Errors at compile time

- Syntax errors: easy to fix
- Library issues
- Cross-compiling
- Compiler warnings
  Always switch this on, and fix or understand them!

Common symptoms

Errors at compile time

- Syntax errors: easy to fix
- Library issues
- Cross-compiling
- Compiler warnings
  Always switch this on, and fix or understand them!

But just because it compiles does not mean it is correct!

Common symptoms

Errors at compile time

- Syntax errors: easy to fix
- Library issues
- Cross-compiling
- Compiler warnings
  Always switch this on, and fix or understand them!

But just because it compiles does not mean it is correct!

Runtime errors

Common symptoms

Errors at compile time

- Syntax errors: easy to fix
- Library issues
- Cross-compiling
- Compiler warnings
  Always switch this on, and fix or understand them!

But just because it compiles does not mean it is correct!

Runtime errors

- Floating point exceptions
- Segmentation fault
- Aborted
- Incorrect output (nans)

Common issues

Arithmetic	corner cases (`sqrt(-0.0)`), infinities
Memory access	Index out of range, uninitialized pointers.
Logic	Infinite loop, corner cases
Misuse	wrong input, ignored error, no initialization
Syntax	wrong operators/arguments
Resource starvation	memory leak, quota overflow
Parallel	race conditions, deadlock

Ways to debug

Ways to debug

- Preemptive:
  - Turn on compiler warnings: fix or understand them!

    ```
    $ gcc/g++ -Wall
    ```

  - Check your assumptions (e.g. use **`assert`**).

Ways to debug

- Preemptive:
  - Turn on compiler warnings: fix or understand them!

    ```
    $ gcc/g++ -Wall
    ```

  - Check your assumptions (e.g. use **`assert`**).
- Inspect the exit code and read the error messages!
- Use a debugger
- Add print statements

Ways to debug

- Preemptive:
  - Turn on compiler warnings: fix or understand them!

    ```
    $ gcc/g++ -Wall
    ```

  - Check your assumptions (e.g. use **`assert`**).
- Inspect the exit code and read the error messages!
- Use a debugger
- Add print statements ←**No way to debug!**

What's wrong with using print statements?

Strategy

What's wrong with using print statements?

Strategy

- Constant cycle:

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

What's wrong with using print statements?

Strategy

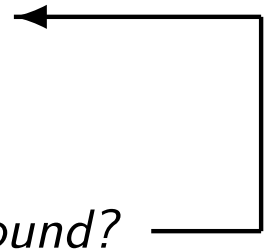
- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output *bug not found?*

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

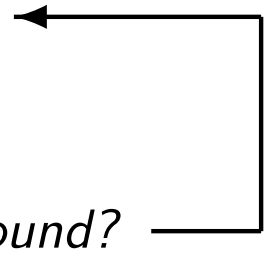
bug not found?



What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

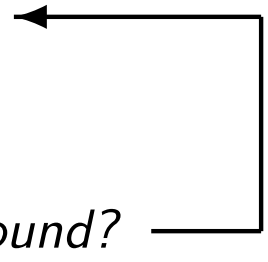


- Removing the extra code after the bug is fixed

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

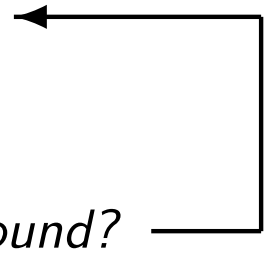


- Removing the extra code after the bug is fixed
- Repeat for each bug

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output



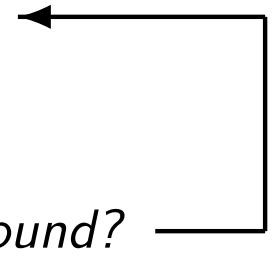
- Removing the extra code after the bug is fixed
- Repeat for each bug

Problems with this approach

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output



- Removing the extra code after the bug is fixed
- Repeat for each bug

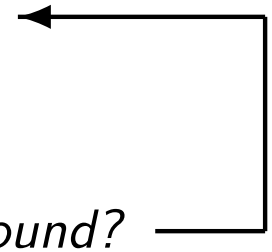
Problems with this approach

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

What's wrong with using print statements?

Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

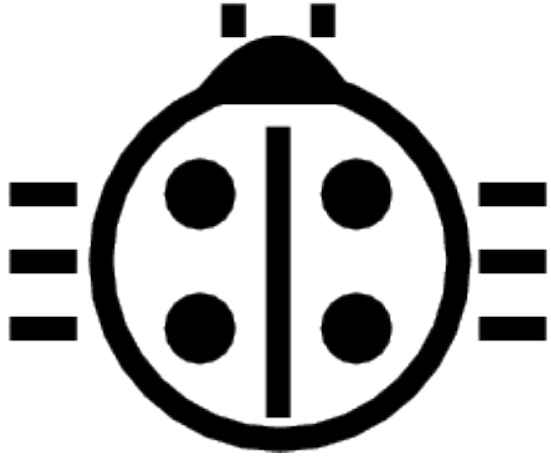


- Removing the extra code after the bug is fixed
- Repeat for each bug

Problems with this approach

- Time consuming
- Error prone
- Changes memory, timing... **There's a better way!**

Symbolic debuggers



Symbolic debuggers

Features

Symbolic debuggers

Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

Symbolic debuggers

Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

Use a graphical debugger or not?

Symbolic debuggers

Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

Use a graphical debugger or not?

- Local work station: graphical is convenient
- Remotely (SciNet): can be slow

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

Symbolic debuggers

Preparing the executable

- Add required compilination flags:
  - `$ gcc/g++/gfortran -g -gstabs`
  - `$ icc/icpc/ifort -g -debug all`
  - `$ nvcc -g -G`
- Optional: switch off optimization `-O0`

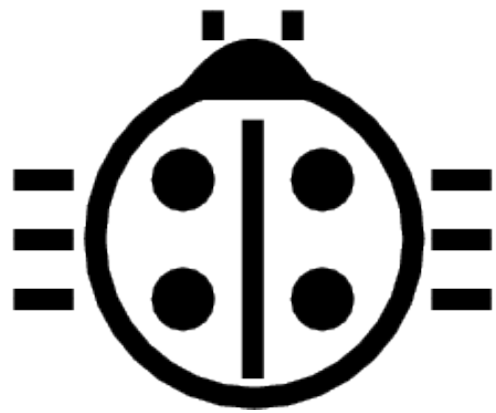
Symbolic debuggers

Preparing the executable

- Add required compilination flags:
  ```
  $ gcc/g++/gfortran -g -gstabs
  $ icc/icpc/ifort -g -debug all
  $ nvcc -g -G
  ```
- Optional: switch off optimization **-O0**

Command-line based symbolic debuggers: gdb

GDB



SciNet
compute • calcul
CANADA

What is GDB?

- Free, GNU license, symbolic debugger.
- Available on many systems.
- Been around for a while, but still developed and up-to-date
- Text based, but has a '-tui' option.

```
$ module load gcc
$ gcc -g -O0 example.c -o example
$ module load gdb
$ gdb -tui example
...
(gdb)_
```

GDB command summary

help	**h**	print description of
run	**r**	run from the start (+args)
backtrace/where	**ba**	function call stack
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
quit	**q**	quit
finish	**fin**	continue until function end
set variable	**set var**	change variable
down	**do**	go to called function
tbreak	**tb**	set temporary breakpoint
until	**unt**	continue until line/function
up	**up**	go to caller
watch	**wa**	stop if variable changes
quit	**q**	quit gdb

GDB basic building blocks

SciNet
compute • calcul
CANADA

GDB building block #1: Inspect crashes

Inspecting core files

GDB building block #1: Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

GDB building block #1: Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

- NOTE: needs max core size set (**`ulimit -c <number>`**)
- gdb reads with **`gdb <executable> <corefile>`**
- it will show you where the program crashed

GDB building block #1: Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

- NOTE: needs max core size set (**`ulimit -c <number>`**)
- gdb reads with **`gdb <executable> <corefile>`**
- it will show you where the program crashed

No core file?

GDB building block #1: Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

- NOTE: needs max core size set (**`ulimit -c <number>`**)
- gdb reads with **`gdb <executable> <corefile>`**
- it will show you where the program crashed

No core file?

- can start gdb as **`gdb <executable>`**
- type **`run`** to start program
- gdb will show you where the program crashed if it does.

GDB Exampe: Code

```
#include <iostream>
#include <cmath>
int main(int argc, char **argv) {
  int nmax; float *squares, sum;
  std::cin >> nmax;
  for (int i=1; i<=nmax; i++) {
    squares[i] = sqrt(i-2);
    sum += squares[i];
  }
  std::cout << sum;
  return 0;
}
```

GDB Example #1

```
$g++ -g -o square square.c
$./square
5000
Segmentation fault
```

GDB Example #1

```
$g++ -g -o square square.c
$./square
5000
Segmentation fault
```

```
$gdb ./square core.12345
Program terminated with signal 11, Segmentation fault.
#0 0x0000000000400855 in main (argc=2,
argv=0x7fff6db1ac18) at square.c:12
12 squares[i] = sqrt( (i-2) );
(gdb)
```

GDB building block #2: Function call stack

Interrupting program

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

GDB building block #2: Function call stack

Interrupting program

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

Stack trace

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

GDB building block #2: Function call stack

Interrupting program

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

Stack trace

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

gdb commands

backtrace	function call stack
continue	continue
down	go to called function
up	go to caller

GDB building block #3: Step through code

Stepping through code

- Line-by-line
- Choose to step into or over functions
- Can show surrounding lines or use **`-tui`**

GDB building block #3: Step through code

Stepping through code

- Line-by-line
- Choose to step into or over functions
- Can show surrounding lines or use **`-tui`**

gdb commands

`list`	list part of code
`next`	continue until next line
`step`	step into function
`finish`	continue until function end
`until`	continue until line/function

GDB building block #4: Automatic interruption

Breakpoints

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

GDB building block #4: Automatic interruption

Breakpoints

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

Related gdb commands

`delete`	unset breakpoint
`condition`	break if condition met
`disable`	disable breakpoint
`enable`	enable breakpoint
`info breakpoints`	list breakpoints
`tbreak`	temporary breakpoint

GDB building block #5: Variables

Checking a variable

- 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 . . . ")

GDB building block #5: Variables

Checking a variable

- 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 . . . ")

gdb commands

`print`	print variable
`display`	print at every prompt
`set variable`	change variable
`watch`	stop if variable changes

GDB Exampe: Code (fixed)

```
#include <iostream>
#include <cmath>
int main(int argc, char **argv) {
  int nmax; float *squares, sum;
  std::cin >> nmax;
  squares = new float [nmax]; //allocate memory
  for (int i=1; i<=nmax; i++) {
    squares[i] = sqrt(i-2);
    sum += squares[i];
  }
  std::cout << sum;
  return 0;
}
```

GDB : Example #2

```
$gdb ./square
(gdb)
(gdb) list 15
10
11 for (int i=1; i<=nmax; i++) {
12 squares[i] = sqrt( (i-2) );
13 sum += squares[i];
14 }
15
16 std::cout << sum;
17 return 0;
18 }
19
(gdb) break 13
Breakpoint 1 at 0x4008ae:  file square.2.c, line 13.
```

GDB : Example #2 - continued

```
(gdb) run
Starting program:  ./square
20
Breakpoint 1, main (argc=1, argv=0x7fffffffdc48) at
square.2.c:13
13 sum += squares[i];
(gdb) print sum
$1 = 5.88011741e-39
(gdb) step
11 for (int i=0; i<nmax; i++) {
(gdb) print sum
$2 = -nan(0x400000)
(gdb) print squares[0]
$3 = -nan(0x400000)
(gdb) quit
```

Graphical symbolic debuggers

SciNet
compute • calcul
CANADA

Graphical symbolic debuggers

Features

- Nice, more intuitive graphical user interface
- Front to command-line based tools: Same concepts
- Need graphics support: X forwarding (or VNC)

Graphical symbolic debuggers

Features

- Nice, more intuitive graphical user interface
- Front to command-line based tools: Same concepts
- Need graphics support: X forwarding (or VNC)

Available on SciNet: ddd and ddt

- ddd

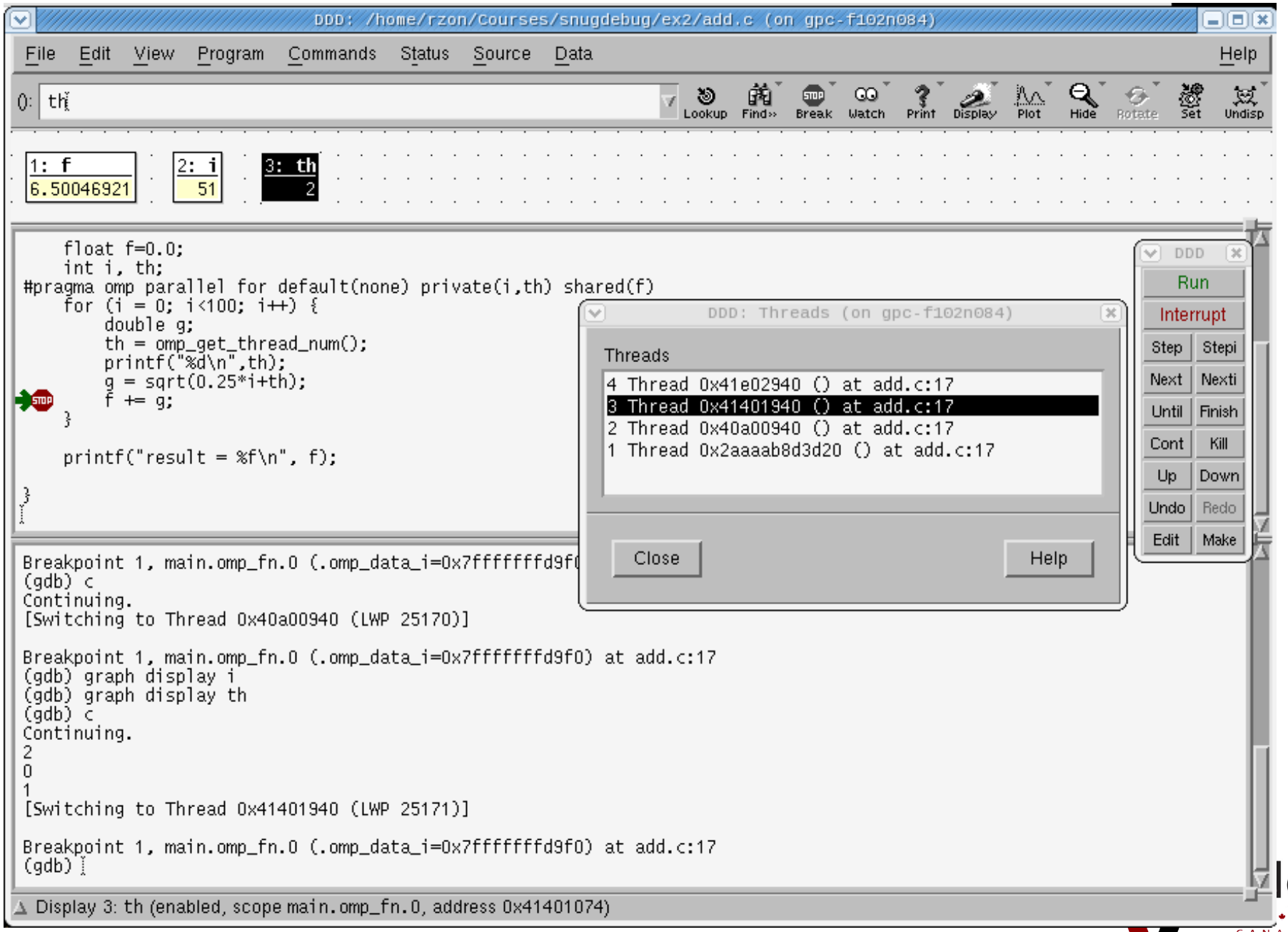
```
$ module load gcc ddd
$ ddd <executable compiled with -g flag>
```

- ddt

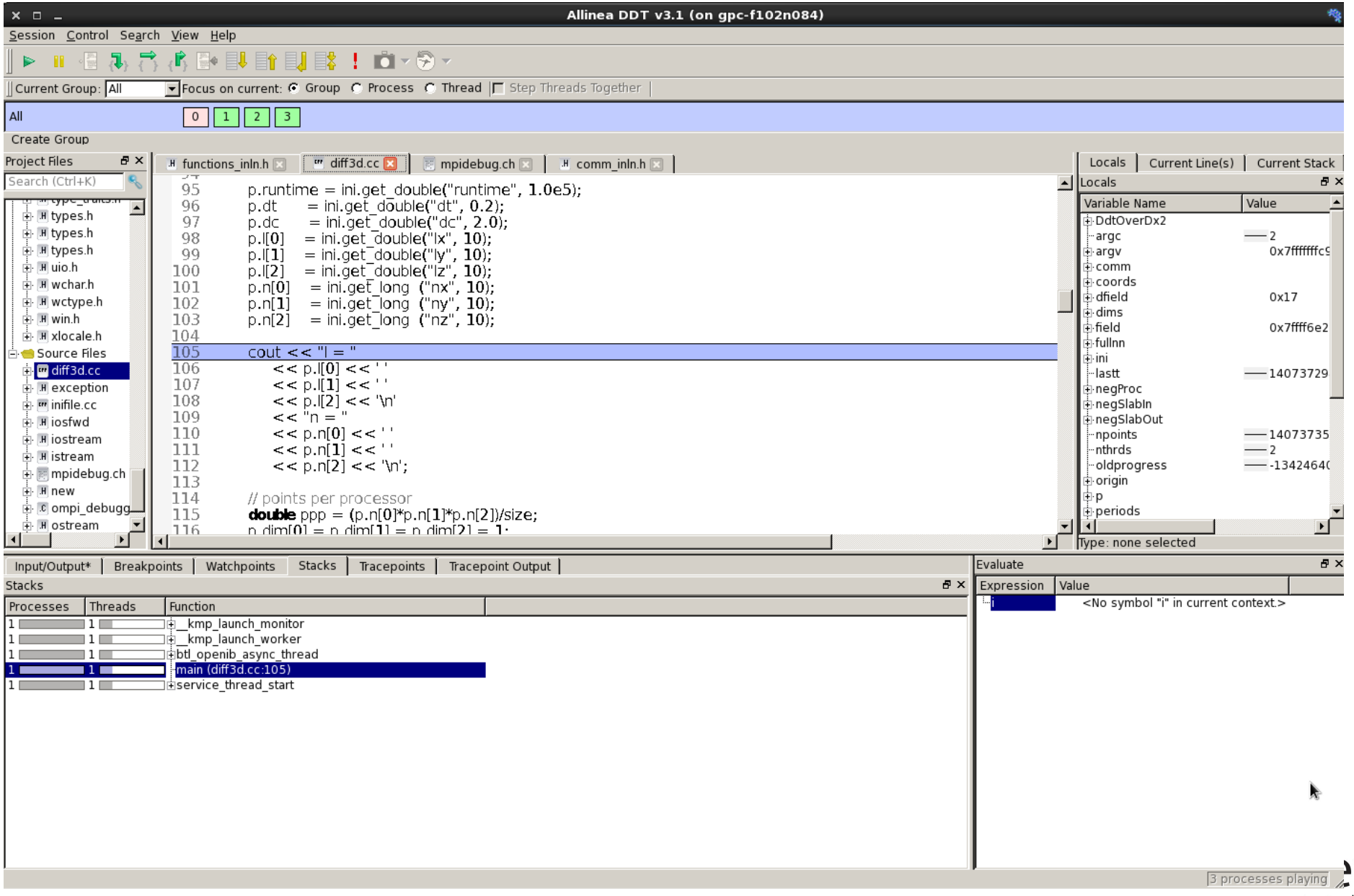
```
$ module load ddt
$ ddt <executable compiled with -g flag>
(more later)
```

SciNet compute • calcul CANADA

Graphical symbolic debuggers - ddd



Graphical symbolic debuggers - ddt



Memory Checking: Valgrind

- Memory errors do not always give segfaults
- Commonly have to go *way* out of bounds to get a segfault.
- Write into other variables - hard to find problem.
- Valgrind - intercepts each memory call and checks them (very thorough but slow).
- Finds illegal accesses, uninitialized values, memory leaks.
- Is typically very verbose.
- If you use external libraries, sometimes false positive

GDB Exampe: Code (fixed?)

```
#include <iostream>
#include <cmath>
int main(int argc, char **argv) {
  int nmax; float *squares, sum;
  std::cin >> nmax;
  squares = new float [nmax]; //allocate memory
  for (int i=1; i<=nmax; i++) {
    squares[i] = sqrt(i); //fixed nan's
    sum += squares[i];
  }
  std::cout << sum;
  return 0;
}
```

Memory Checking: Valgrind

valgrind --tool=memcheck ./square

```
==31550== Invalid write of size 4
==31550==    at 0x4008A5: main (square.c:8)
==31550==  Address 0x4c3b090 is 0 bytes after a block of size 80 alloc'd
==31550==    at 0x4A07152: operator new[](unsigned long) (vg_replace_mallo
==31550==    by 0x400875: main (square..c:6)
==31550==
==31550== Invalid read of size 4
==31550==    at 0x4008B6: main (square.c:9)
==31550==  Address 0x4c3b090 is 0 bytes after a block of size 80 alloc'd
==31550==    at 0x4A07152: operator new[](unsigned long) (vg_replace_mallo
==31550==    by 0x400875: main (square.c:6)
```

Memory Checking: Valgrind

valgrind --tool=memcheck ./square

```
==31550== Invalid write of size 4
==31550==    at 0x4008A5: main (square.c:8)
==31550==  Address 0x4c3b090 is 0 bytes after a block of size 80 alloc'd
==31550==    at 0x4A07152: operator new[](unsigned long) (vg_replace_mallo
==31550==    by 0x400875: main (square..c:6)
==31550==
==31550== Invalid read of size 4
==31550==    at 0x4008B6: main (square.c:9)
==31550==  Address 0x4c3b090 is 0 bytes after a block of size 80 alloc'd
==31550==    at 0x4A07152: operator new[](unsigned long) (vg_replace_mallo
==31550==    by 0x400875: main (square.c:6)
```

Error:

i index from 1 to nmax

GDB Exampe: Code (fixed?)

```
#include <iostream>
#include <cmath>
int main(int argc, char **argv) {
  int nmax; float *squares, sum;
  std::cin >> nmax;
  squares = new float [nmax]; //allocate memory
  for (int i=0; i<nmax; i++) //fixed i index {
    squares[i] = sqrt(i); //fixed nan's
    sum += squares[i];
  }
  std::cout << sum;
  return 0;
}
```

Memory Checking: Valgrind

```
==31550== Conditional jump or move depends on uninitialised value(s)
==31550==    at 0x3A41243696: __mpn_extract_double (in /lib64/libc-2.12.so
==31550==    by 0x3A4124A4BD: __printf_fp (in /lib64/libc-2.12.so)
==31550==    by 0x3A41245B9F: vfprintf (in /lib64/libc-2.12.so)
==31550==    by 0x3A4126FA51: vsnprintf (in /lib64/libc-2.12.so)
==31550==    by 0x3A47E7EB4E: ??? (in /usr/lib64/libstdc++.so.6.0.13)
==31550==    by 0x3A47E80F22: std::ostreambuf_iterator<char, std::char_tra
==31550==    by 0x3A47E81248: std::num_put<char, std::ostreambuf_iterator<
==31550==    by 0x3A47E9487E: std::ostream& std::ostream::_M_insert<double
==31550==    by 0x4008E7: main (square.c:11)

==31550== Use of uninitialised value of size 8
```

Memory Checking: Valgrind

```
==31550== Conditional jump or move depends on uninitialised value(s)
==31550==    at 0x3A41243696: __mpn_extract_double (in /lib64/libc-2.12.so
==31550==    by 0x3A4124A4BD: __printf_fp (in /lib64/libc-2.12.so)
==31550==    by 0x3A41245B9F: vfprintf (in /lib64/libc-2.12.so)
==31550==    by 0x3A4126FA51: vsnprintf (in /lib64/libc-2.12.so)
==31550==    by 0x3A47E7EB4E: ??? (in /usr/lib64/libstdc++.so.6.0.13)
==31550==    by 0x3A47E80F22: std::ostreambuf_iterator<char, std::char_tra
==31550==    by 0x3A47E81248: std::num_put<char, std::ostreambuf_iterator<
==31550==    by 0x3A47E9487E: std::ostream& std::ostream::_M_insert<double
==31550==    by 0x4008E7: main (square.c:11)

==31550== Use of uninitialised value of size 8
```

Error:

variable “sum” never initialized

GDB Exampe: Code (fixed?)

```
#include <iostream>
#include <cmath>
int main(int argc, char **argv) {
  int nmax; float *squares, sum(0); //init sum
  std::cin >> nmax;
  squares = new float [nmax]; //allocate memory
  for (int i=0; i<nmax; i++)//fixed i index {
    squares[i] = sqrt(i); //fixed nan's
    sum += squares[i];
  }
  std::cout << sum;
  return 0;
}
```

Memory Checking: Valgrind

```
==31550== HEAP SUMMARY:
==31550==     in use at exit: 80 bytes in 1 blocks
==31550==   total heap usage: 1 allocs, 0 frees, 80 bytes allocated
==31550==
==31550== LEAK SUMMARY:
==31550==    definitely lost: 80 bytes in 1 blocks
==31550==    indirectly lost: 0 bytes in 0 blocks
==31550==      possibly lost: 0 bytes in 0 blocks
==31550==    still reachable: 0 bytes in 0 blocks
==31550==         suppressed: 0 bytes in 0 blocks

==31550== ERROR SUMMARY: 204 errors from 113 contexts (suppressed: 6 from
```

Memory Checking: Valgrind

```
==31550== HEAP SUMMARY:
==31550==     in use at exit: 80 bytes in 1 blocks
==31550==   total heap usage: 1 allocs, 0 frees, 80 bytes allocated
==31550==
==31550== LEAK SUMMARY:
==31550==    definitely lost: 80 bytes in 1 blocks
==31550==    indirectly lost: 0 bytes in 0 blocks
==31550==      possibly lost: 0 bytes in 0 blocks
==31550==    still reachable: 0 bytes in 0 blocks
==31550==         suppressed: 0 bytes in 0 blocks

==31550== ERROR SUMMARY: 204 errors from 113 contexts (suppressed: 6 from
```

Error:

forgot to free dynamic memory squares

GDB Exampe: Code (fixed?)

```
#include <iostream>
#include <cmath>
int main(int argc, char **argv) {
  int nmax; float *squares, sum(0); //init sum
  std::cin >> nmax;
  squares = new float [nmax]; //allocate memory
  for (int i=0; i<nmax; i++) //fixed i index{
    squares[i] = sqrt(i); //fixed nan's
    sum += squares[i];
  }
  std::cout << sum;
  delete [] squares; //deallocate memory
  return 0;
}
```

Memory Checking: Valgrind

```
$  valgrind --tool=memcheck ./square
==31707== Memcheck, a memory error detector
==31707== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==31707== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright inf
==31707== Command: ./square
==31707==
20
57.1938
==31707==
==31707== HEAP SUMMARY:
==31707==     in use at exit: 0 bytes in 0 blocks
==31707==   total heap usage: 1 allocs, 1 frees, 80 bytes allocated
==31707==
==31707== All heap blocks were freed -- no leaks are possible
==31707==
==31707== For counts of detected and suppressed errors, rerun with: -v
==31707== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```