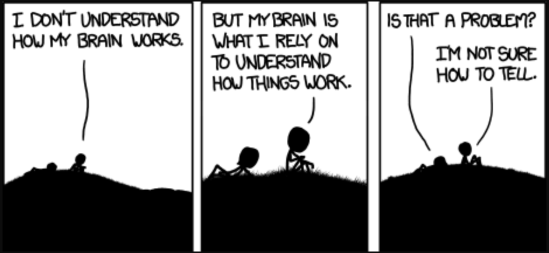
Debugging

Ramses van Zon

PHY1610H Winter 2026

What if your program or test isn't running correctly. . .

- Nonsense. All programs execute "correctly".
- 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.


I DON'T UNDERSTAND HOW MY BRAIN WORKS.
BUT MY BRAIN IS WHAT I RELY ON TO UNDERSTAND HOW THINGS WORK.
IS THAT A PROBLEM?
I'M NOT SURE HOW TO TELL.

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

Debugger: program to help detect errors in other programs.

Some 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

Debugging workflows

- As soon as you are convinced there is a real problem, create the simplest situation in which it repeatedly occurs.
- Take a scientific approach: 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.

Ways to debug

To figure out what is going wrong, and where in the code, we can

1. Put strategic print statements in the code.
2. Use a debugger.

We don't like the first option.

What's wrong with using print statements?

Uses the following strategy

- Constant cycle:
  - strategically add print statements
  - compile
  - run
  - analyze output
  - repeat
- Removing the extra code after the bug is fixed
- Repeat for each bug

Problems with this approach

A bug is always unexpected, so you don't know where to put those strategic print statements.

As a result, this approach:

- is time consuming
- is error prone
- is confusing as print output might not appear when you think
- changes memory layout, output format, timing, etc.

There's a better way!

Debuggers

are programs that can show what happens in a program at runtime.

Features

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

Should you use a graphical/IDE debugger?

- Local work station: graphical/IDE is convenient
- Remotely (SciNet): can be slow or hard to set up.
- In any case, graphical and text-based debuggers use the same concepts.

Debuggers

Preparing the executable for debugging

- Add required compilation flags, `-g`

  (both in compiling and linking!)
- Recommended: switch off optimization `-O0`

Command-line based symbolic debugger: gdb

- Free, GNU license, symbolic debugger.
- Available on many systems.
- Been around for a while, but still developed and up-to-date
- Command-line based, does not show code listing by default, unless you use the `-tui` option.

Example

Consider this code:

```
// crashex.cpp
#include <print>
void handle_command_line(int argc, char* argv[]){
  if (argv[1][0] == '-' && argv[1][1] == 'h') {
    // print help
    std::println("Usage:  crashex [-h]\n");
  } else {
    // ....
  }
}
int main(int argc, char** argv) {
  handle_command_line(argc,argv);
  // ...
}
```

which we compile on Teach with:

```
$ module load gcc/14.3
$ g++ -std=c++23 -O0 -g crashex.cpp -o crashex
```

When run, it shows the following:

```
$ ./crashex -h
Usage:  crashex [-h]
```

```
$ ./crashex
Segmentation fault (core dumped)
```

The first invocation works, but the second fails.

Why?

Crash inspection

```
$ ./crashex
Segmentation fault (core dumped)
```

We want to solve this segmentation fault.

- A segmentation fault means that your application is trying to access data at an invalid memory location.
- When the operating system detects the invalid memory location, it kills the application in that case and produces a core dump.
- The core contains the process's memory state, call stack, and failure mode at the moment of the crash, like a "black box".

Missing the core file?

Core size limit

If the error message did not say `core dumped`, you need to set the limit

```
# ulimit -c unlimited
```

Still no core file?

Core dump files used to always appear in the current directory with a name starting with `core` followed by the process ID.

Modern Linux distribution, handle core dumps in a variety of ways, but the original way is most convenient for debugging.

For Ubuntu and RedHat, the old default behaviour can be restored with the following command:

```
$ sudo sysctl -w kernel.core_pattern=core.%p
```

Inspecting the crash with gdb

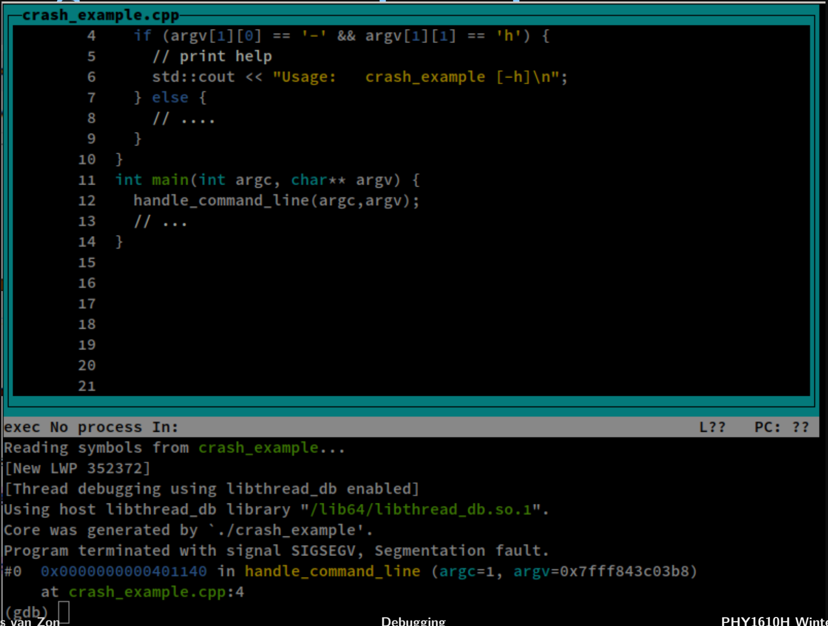
To inspect the crash, use the gdb command followed by the name of the application and the name of the core file, e.g. on Teach:

```
$ gdb ./crashex core.teach-login01.crashex.203739
```

```
GNU gdb (Gentoo 13.2 vanilla) 13.2
<A header with general gdb information>
Reading symbols from ./crashex...
[New LWP 203739]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/cvmfs/soft.computecanada.ca/gentoo/2023/x86-64-v3/usr/lib64/libthread_d
Core was generated by `./crashex`.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000559697c9f1a7 in handle_command_line (argc=1, argv=0x7ffda30f8c98)
    at crashex.cpp:4
4       if (argv[1][0] == '-' && argv[1][1] == 'h') {
(gdb)
```

This shows the error occured at line 4!

Tip: start gdb with -tui -quiet options



```
crash_example.cpp
  if (argv[1][0] == '-' && argv[1][1] == 'h') {
    // print help
    std::cout << "Usage:   crash_example [-h]\n";
  } else {
    // ....
  }
}
int main(int argc, char** argv) {
  handle_command_line(argc,argv);
  // ...
}
```

```
exec No process In:                                                L??   PC: ??
Reading symbols from crash_example...
[New LWP 352372]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Core was generated by `./crash_example'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000000401140 in handle_command_line (argc=1, argv=0x7fff843c03b8)
    at crash_example.cpp:4
(gdb) 
```

Function Call Stack:

One of the commands availble at the gdb prompt is `backtrace`:

```
(gdb) backtrace
#0  0x0000559697c9f1a7 in handle_command_line (argc=1, argv=0x7ffda30f8c98)
    at crashex.cpp:4
#1  0x0000559697c9f204 in main (argc=1, argv=0x7ffda30f8c98)
    at crashex.cpp:12
```

This shows the line of the crash again, but also how the code got there, i.e., from line 12 in the `main` function in the file `crashex.cpp`.

Note:

- Hexadecimal numbers at the beginning of the lines refer to positions in the executable.
- The backtrace also shows the values of the arguments, with pointers printed as hexadecimal memory addresses.

Checking Variables:

The printed values of the arguments given by the backtrace do not help us.

But it would be helpful to known the values that occur in line 4 where the crash happens.

The gdb command for this is `print`, which makes print statements in the code unncessary for debugging.

Let's try it here:

```
(gdb) print argv[1][0]
Cannot access memory at address 0x0
```

This should be a surprise; where does an address 0x0 come from?

Let's try this:

```
(gdb) print argv[1]
0x0
```

This is a solid clue on what the bug is: we are dereferencing a null pointer.

The bug is that the code did not check if a first argument is actually present.

Stepping Through Code

The debugger can also execute the code line-by-line, but the code has already crashed.

So we exit the debugger with the `quit` command, and start it again without the core file:

```
$ gdb -tui -quiet ./crashex
GNU gdb (GDB) 13.2
Reading symbols from ./crashex...
(gdb)
```

It shows the code, but nothing has started running yet.

We can now type `run`, which would run the code, but it would just lead to the crash again.

Instead, we want to pause at the start of `main`, which the `start` command does:

```
(gdb) start
Temporary breakpoint 1 at 0x11f3: file crashex.cpp, line 12.
Starting program: /home/rzon/crashex
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Temporary breakpoint 1, main (argc=1, argv=0x7fffffffc3c8)
 at crashex.cpp:12
12 handle_command_line(argc,argv);
```

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Stepping Through Code (cont.)

The executation is now paused at line 12, which has not yet been executed.

When execution is paused, we can print expressions, but we can now also step forward in the code.

For this, there are two commands:

- `next` executes this whole line, which in this case would again lead to the crash.
- `step` will step into any function calls on the line, i.e., it will pause at the first line of the first function call in the line.

Let's try `step` here:

```
(gdb) step
handle_command_line (argc=1, argv=0x7ffffffc3c8) at crashex.cpp:4
4 if (argv[1][0] == '-' && argv[1][1] == 'h') {
```

which is indeed the first line of the function `handle_command_line` that was called from line 12.

Whenever the code is paused, we can print variables and expressions, but this time, before the crash has occured.

Breakpoints

If you have a code where a crash or bug occurs only after the process has run for a while, having to step through the code from the start would be very inefficient.

Instead, you can setup an automated interruption called a break point at either a line or code or a function, and run the code until just before that point is executed.

The `break` command sets this up. E.g.

```
(gdb) break 4
Breakpoint 2 at 0x55555555519c: file crashex.cpp, line 4.
```

```
(gdb) break handle_command_line
Note: breakpoint 2 also set at pc 0x55555555519c.
Breakpoint 3 at 0x55555555519c: file crashex.cpp, line 4.
```

You can unset a breakpoint with the `delete` command, e.g. `delete 3`.

Breakpoints (cont.)

With a breakpoint set, we can now run the code with `run`.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/rzon/crashex
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 2, handle_command_line (argc=1, argv=0x7fffffffc3c8)
    at crashex.cpp:4
4     if (argv[1][0] == '-' && argv[1][1] == 'h') {
```

As this example shows, the `run` command will ask you if you want to restart the application if it was already running.

If instead of starting from the start, you wanted to continue from where the code was, use the `continue` command instead.

Setting Variables

When the program is paused in gdb, you can change the values of variables as well with `set`.

This will change the execution of the code and can be useful for experimentation, but keep in mind that this also makes the debugging process harder to reproduce.

For example, we could set the value of `argv[1]` to something valid:

```
(gdb) print argv[1]
$1 = 0x0
```

```
(gdb) set argv[1] = "-h"
```

```
(gdb) p argv[1]
$2 = 0x55555556aeb0 "-h"
```

```
(gdb) continue
Continuing.
Usage:  crashex [-h]
[Inferior 1 (process 215420) exited normally]
```

Note: Here "Inferior" merely refers to the fact that the code was run under gdb.

GDB command summary

`help`	`h`	print description of command
`run`	`r`	run from beginning (+args)
`start`	`start`	run from main
`backtrace`	`ba`	function call stack
`break`	`b`	set breakpoint
`delete`	`d`	delete breakpoint
`continue`	`c`	continue
`list`	`l`	print part of the code
`step`	`s`	step into function
`next`	`n`	continue until next line
`print`	`p`	print variable
`display`	`disp`	print variable at every prompt
`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

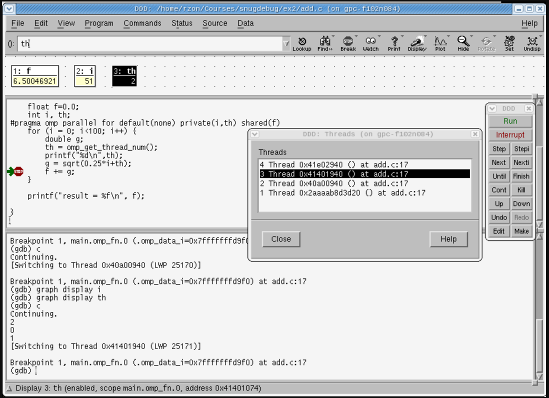
Graphical debuggers

DDD: free, bit old, can do serial and threaded debugging.

```
module load ddd
```

DDT: commercial, on SciNet, part of “Linaro Forge”; good for parallel debugging.

```
module load ddt-cpu
```



DDD: /home/rzon/Courses/snugdebug/ex2/add.c (on gpc-f102n084)
File Edit View Program Commands Status Source Data Help
(): th
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
1: f            2: i    3: th
6.50046921      51      2
```

```
    float f=0.0;
    int i, th;
#pragma omp parallel for default(none) private(i,th) shared(f)
    for (i = 0; i<100; i++) {
        double g;
        th = omp_get_thread_num();
        printf("%d\n",th);
        g = sqrt(0.25*i+th);
        f += g;
    }

    printf("result = %f\n", f);
}
```

DDD: Threads (on gpc-f102n084)

```
Threads
4 Thread 0x41e02940 () at add.c:17
3 Thread 0x41401940 () at add.c:17
2 Thread 0x40a00940 () at add.c:17
1 Thread 0x2aaaab8d3d20 () at add.c:17
```

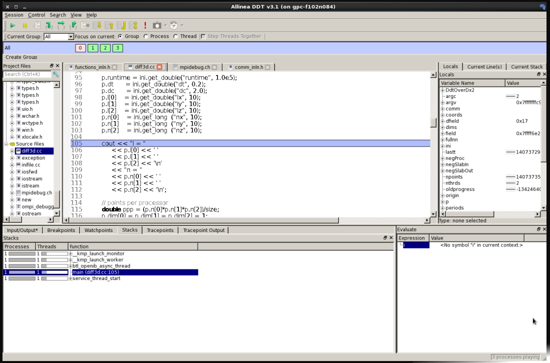
Close Help

```
Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0
(gdb) c
Continuing.
[Switching to Thread 0x40a00940 (LWP 25170)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0) at add.c:17
(gdb) graph display i
(gdb) graph display th
(gdb) c
Continuing.
2
0
1
[Switching to Thread 0x41401940 (LWP 25171)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0) at add.c:17
(gdb)
```

Display 3: th (enabled, scope main.omp_fn.0, address 0x41401074)



Allinea DDT v3.1 (on gpc-f102n084)
Session Control Search View Help
Current Group: All Focus on current: Group Process Thread Step Threads Together
All 0 1 2 3
Create Group
Project Files
functions_inln.h diff3d.cc mpidebug.ch comm_inln.h
Locals Current Line(s) Current Stack
Input/Output* Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output
Evaluate
3 processes playing

```
p.runtime = ini.get_double("runtime", 1.0e5);
p.dt      = ini.get_double("dt", 0.2);
p.dc      = ini.get_double("dc", 2.0);
p.l[0]    = ini.get_double("lx", 10);
p.l[1]    = ini.get_double("ly", 10);
p.l[2]    = ini.get_double("lz", 10);
p.n[0]    = ini.get_long  ("nx", 10);
p.n[1]    = ini.get_long  ("ny", 10);
p.n[2]    = ini.get_long  ("nz", 10);

cout << "l = "
    << p.l[0] << ' '
    << p.l[1] << ' '
    << p.l[2] << '\n'
    << "n = "
    << p.n[0] << ' '
    << p.n[1] << ' '
    << p.n[2] << '\n';

// points per processor
double ppp = (p.n[0]*p.n[1]*p.n[2])/size;
p.dim[0] = p.dim[1] = p.dim[2] = 1;
```

```
Processes  Threads  Function
1          1        __kmp_launch_monitor
1          1        __kmp_launch_worker
1          1        btl_openib_async_thread
1          1        main (diff3d.cc:105)
1          1        service_thread_start
```

```
Expression  Value
i           <No symbol "i" in current context.>
```

Tips to avoid debugging

- Write better code.
  - simple, clear, straightfoward code.
  - modularity (avoid global variables and 10,000 line functions).
  - avoid "cute tricks".
- Don't write code, use existing libraries.
- Write (simple) tests for each module.
- Use version control and small commits.
- Switch on the warnings, and understand them all, or better, fix them.

  Tip: use `-Wall -Werror -Wfatal-errors` to stop at the first warning.

- Use defensive programming:

  Check arguments, use assert (which can be switched of with `-DNDEBUG` compilation flag)
  E.g.:

```
#include <cassert>
#include <cmath>
double mysqrt(double x) {
    assert(x>=0);
    return sqrt(x);
}
```