TUTORIAL: INTRODUCTION TO VALGRIND

RODRIGO RANDEL
RODRIGO.RANDEL@GERAD.CA

GERAD

POLYTECHNIQUE
MONTRÉAL

UT TENSIO SIC VIS

LE GÉNIE
EN PREMIÈRE CLASSE

REFERENCE

This presentations was highly inspired by Valgrind Documentation

REPOSITORY WITH EXAMPLES

https://github.com/rodrigorandel/introduction_to_valgrind

WHAT IS CODE DEBUGGING AND PROFILING?

MOTIVATION

- After developing your code, you want make sure that:
  - It is free of bugs
  - Faster as possible

VALGRIND TOOLS THAT CAN AUTOMATICALLY DETECT MANY MEMORY MANAGEMENT AND THREADING BUGS, AND PROFILE YOUR PROGRAMS IN DETAIL

AN OVERVIEW OF VALGRIND

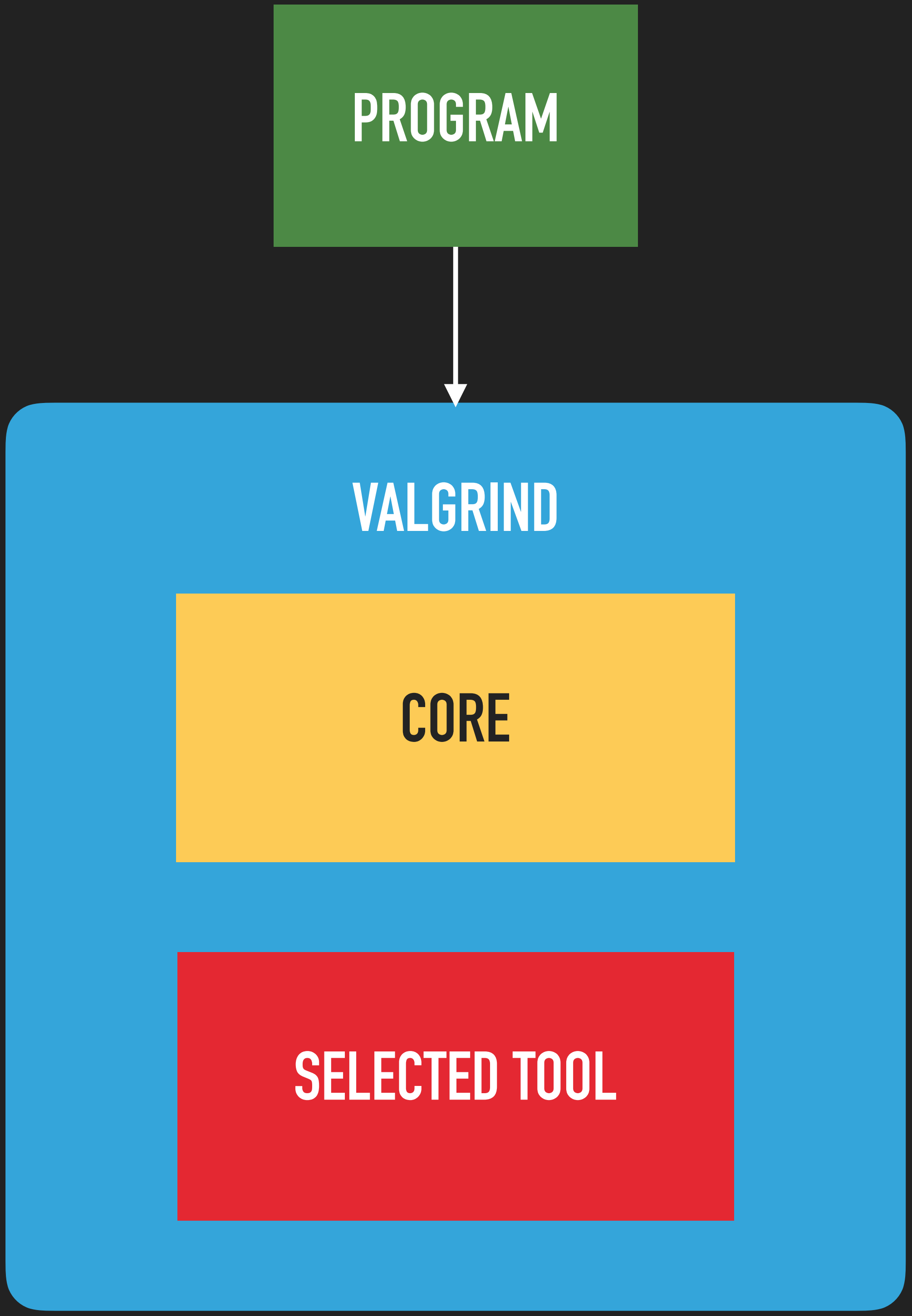
- Valgrind is an instrumentation framework for building dynamic analysis tools.
- Valgrind's architecture is modular, so new tools can be created easily
- Valgrind is designed to be as non-intrusive as possible. It works directly with existing executables

AN OVERVIEW OF VALGRIND

- The Valgrind distribution includes the following debugging and profiling tools:
  - **Memcheck** is a memory error detector. It helps you make your programs more correct (particularly those written in C and C++).
  - **Callgrind** is a call-graph generating cache profiler. It helps identifying the bottleneck of your program.
  - **Massif** is a heap profiler. It helps you make your programs use less memory.
  - **Cachegrind** is a cache and branch-prediction profiler. It helps you make your programs run faster.
  - **Helgrind** is a thread error detector. It helps you make your multi-threaded programs more correct.
  - **DRD** is also a thread error detector. It is similar to Helgrind but uses different analysis techniques and so may find different problems.

HOW VALGRIND WORKS

- Your **program** runs on a synthetic CPU provided by the **Valgrind core**:
  - **When the code is executed for the first time, the core hands the code to the selected tool.**
  - **The tool adds its own instrumentation code**
  - **Hands the result back to the core**
  - **The core coordinates the continued execution of this instrumented code.**



HOW VALGRIND WORKS

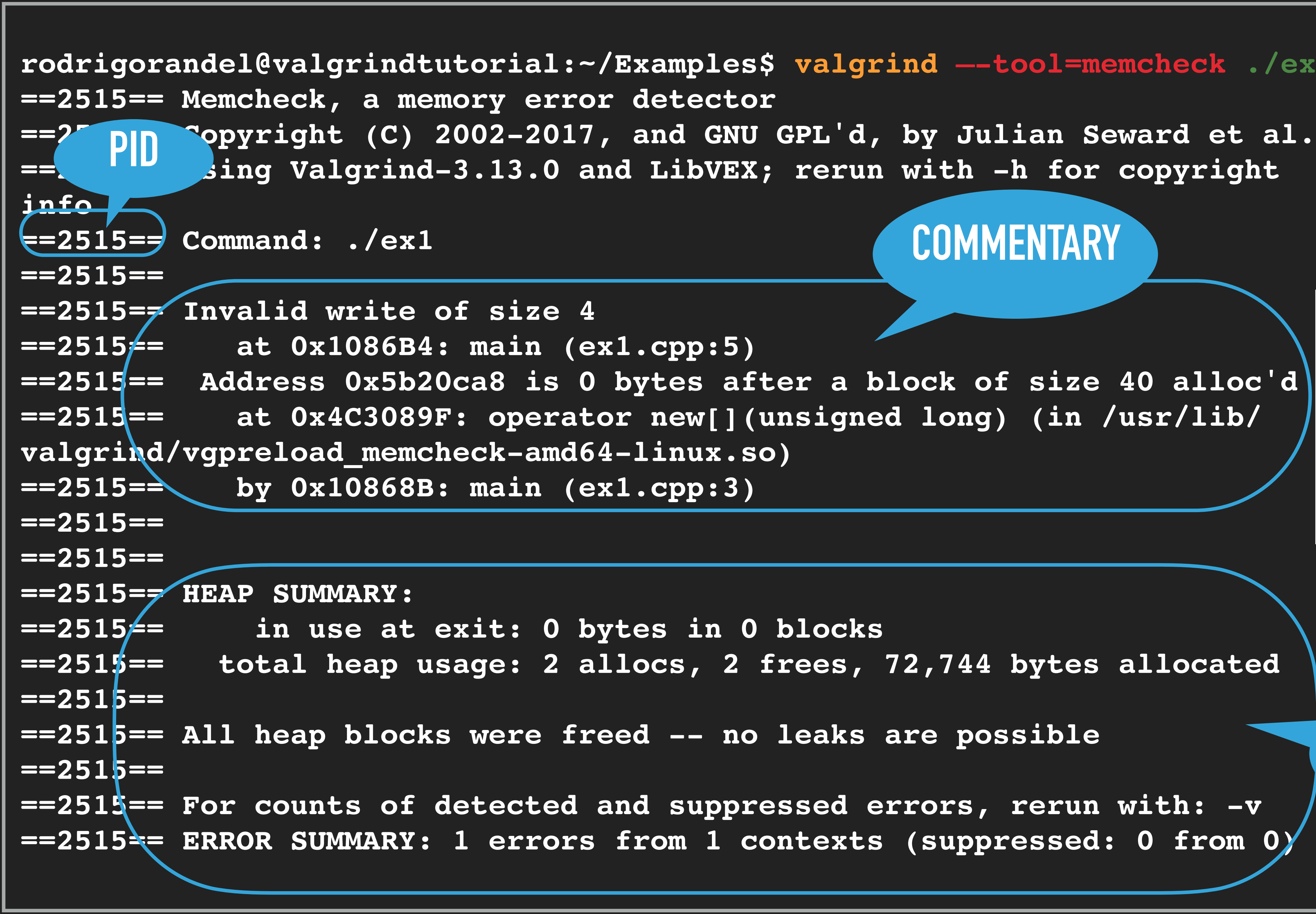
- No need to recompile, relink, or otherwise modify the program to be checked

```
▶ valgrind [valgrind-options] your-prog [your-prog-options]
```

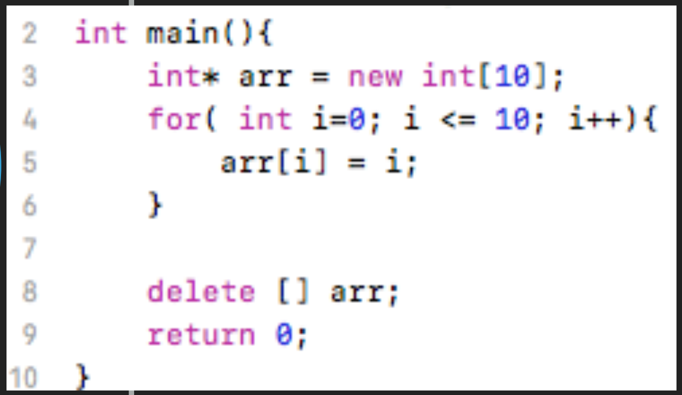
- The most important option is --tool which dictates which Valgrind tool to run.

```
▶ valgrind -tool=memcheck  ls -l
```

UNDERSTANDING VALGRIND'S OUTPUT (MEMCHECK)



```
rodrigorandel@valgrindtutorial:~/Examples$ valgrind --tool=memcheck ./ex1
==2515== Memcheck, a memory error detector
==2515== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2515== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==2515== Command: ./ex1
==2515==
==2515== Invalid write of size 4
==2515==    at 0x1086B4: main (ex1.cpp:5)
==2515==  Address 0x5b20ca8 is 0 bytes after a block of size 40 alloc'd
==2515==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/
valgrind/vgpreload_memcheck-amd64-linux.so)
==2515==    by 0x10868B: main (ex1.cpp:3)
==2515==
==2515==
==2515== HEAP SUMMARY:
==2515==     in use at exit: 0 bytes in 0 blocks
==2515==   total heap usage: 2 allocs, 2 frees, 72,744 bytes allocated
==2515==
==2515== All heap blocks were freed -- no leaks are possible
==2515==
==2515== For counts of detected and suppressed errors, rerun with: -v
==2515== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```



```
int main(){
    int* arr = new int[10];
    for( int i=0; i <= 10; i++){
        arr[i] = i;
    }

    delete [] arr;
    return 0;
}
```

VALGRIND CORE (BASIC) OPTIONS

- --tool=<toolname> [default: memcheck]

  Run the Valgrind tool called toolname
- -h --help

  Show help for all options, both for the core and for the selected tool.
- -q --quiet

  Run silently, and only print error messages.
- -v --verbose

  Gives extra information on various aspects of your program
- --time-stamp=<yes|no> [default: no]

  Each message is preceded with an indication of the elapsed wallclock time since startup
- --log-file=<filename>

  Specifies that Valgrind should send all of its messages to the specified file

DEMO 1: EX1.CPP

REVIEW FROM DEMO 1

- **When using Memcheck with C/C++**
  - **Compile with debugging info (-g for C/C++).**
    - Without debugging info, the best Valgrind tools will be able to do is guess which function a particular piece of code belongs to, which makes both error messages and profiling output nearly useless.
  - **Be careful when compiling with -O1 (and above)**
    - The best solution would be is to turn off optimisation altogether. Since this often makes things unmanageably slow, a reasonable compromise is to use -O1. On rare occasions, compiler optimisations (at -O2 and above, and sometimes -O1) have been observed to generate code which fools Memcheck.

SUPPRESSING ERRORS

SUPPRESSING ERRORS

- The error-checking tools detect numerous problems in the system libraries, such as:
  - C library, which come preinstalled with your OS
  - Third-party software
- You can't easily fix these, but you don't want to see these errors (and yes, there are many!)
- Valgrind reads a list of errors to suppress at startup - default suppression file is created during installation (demo with -v).

WRITING SUPPRESSING ERRORS (GUIDE)

- Asking valgrind to generates suppressions automatically.

```
--gen-suppressions=<yes|no|all> [default: no]

--suppressions=file
```

TEMPLATE

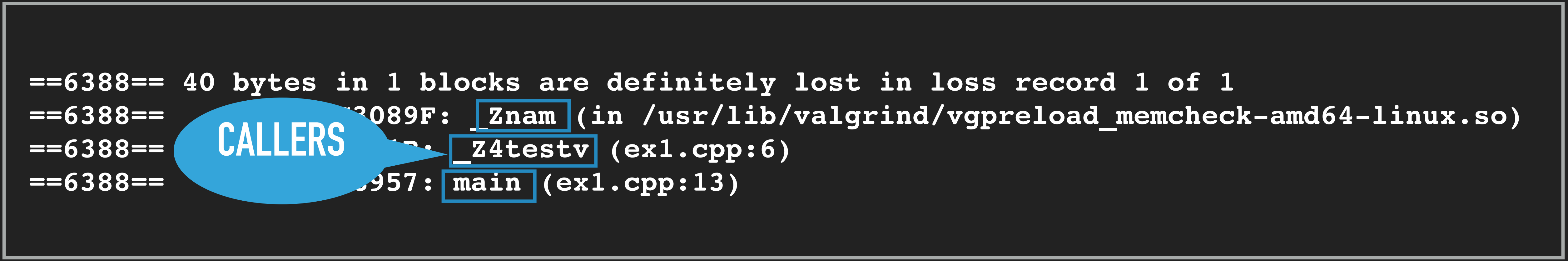
```
{
     name_of_suppression
     tool_name:supp_kind
     (optional extra info for some suppression types)
     caller0 name, or /name/of/so/file.so
     caller1 name, or ditto
     (optionally: caller2 name)
     (optionally: caller3 name)
}
```

WRITING SUPPRESSING ERRORS (GUIDE)

- Write your own suppression file:
  - Use the --demangle=no option to get the mangled names in your error messages

```
==6386== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6386==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/valgrind/
vgpreload_memcheck-amd64-linux.so)
==6386==    by 0x10891B: test() (ex1.cpp:6)
==6386==    by 0x108957: main (ex1.cpp:13)
```

↓



```
==6388== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6388==             3089F: _Znam (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6388==                 B: _Z4testv (ex1.cpp:6)
==6388==              957: main (ex1.cpp:13)
```

DEMO 2: EX2.CPP

DEBUGGING YOUR PROGRAM USING VALGRIND

VALGRIND GDBSERVER AND GDB

- Recall: Your program runs in a synthetic CPU provided by Valgrind
  - A debugger cannot debug your program when it runs on Valgrind.
- **Valgrind gdbserver** provides a fully debuggable program under Valgrind
  - GDB also provides an interactive usage of Valgrind core or tool functionalities,

VALGRIND GDBSERVER (BASICS) OPTIONS

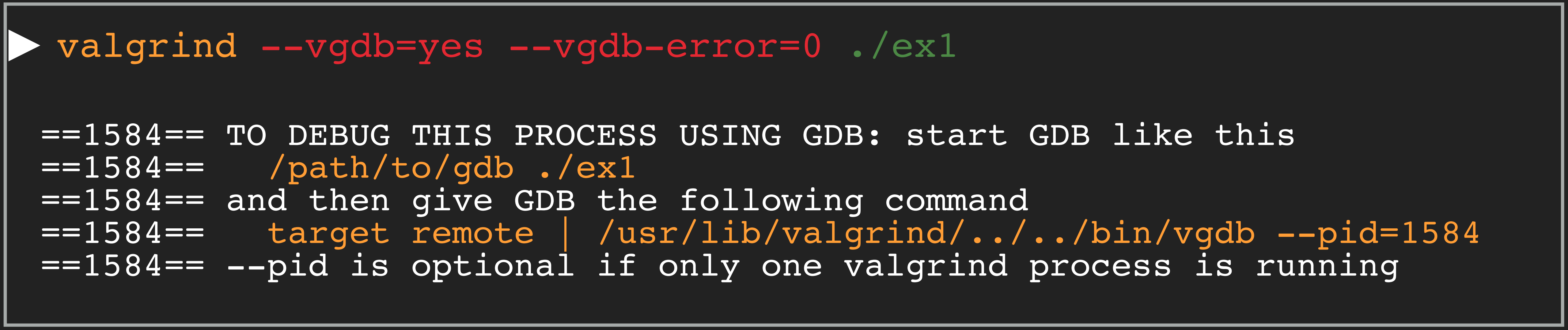
- --vgdb=<no|yes|full> [default: yes]

  Provide "gdbserver" functionality when --vgdb=yes or --vgdb=full is specified

  --vgdb=full provides more precise breakpoints (incurs performance overheads)

- --vgdb-error=<number> [default: 999999999]

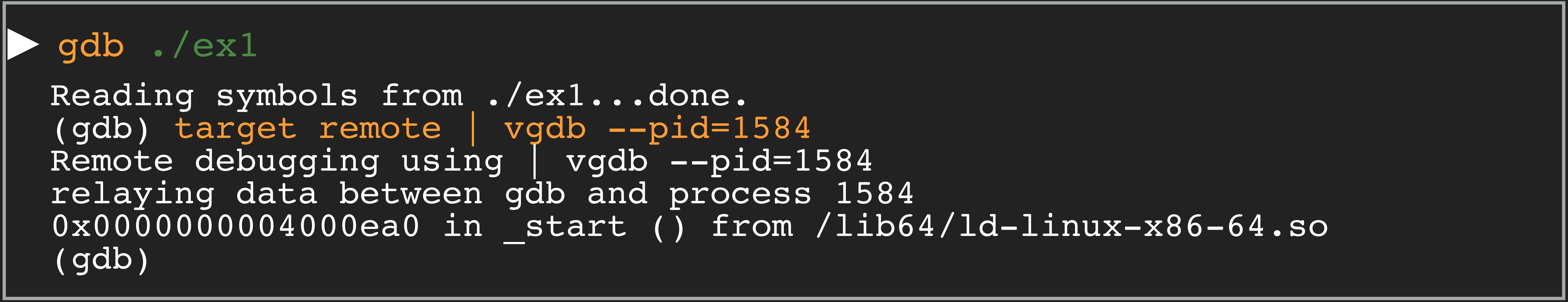
  Tools that report errors will wait for "number" errors to be reported before freezing the program and waiting for you to connect with GDB



```
▶ valgrind --vgdb=yes --vgdb-error=0 ./ex1

 ==1584== TO DEBUG THIS PROCESS USING GDB: start GDB like this
 ==1584==   /path/to/gdb ./ex1
 ==1584== and then give GDB the following command
 ==1584==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=1584
 ==1584== --pid is optional if only one valgrind process is running
```

DEBUGGING USING VALGRIND



```
▶ gdb ./ex1
  Reading symbols from ./ex1...done.
  (gdb) target remote | vgdb --pid=1584
  Remote debugging using | vgdb --pid=1584
  relaying data between gdb and process 1584
  0x0000000004000ea0 in _start () from /lib64/ld-linux-x86-64.so
  (gdb)
```

- Breakpoints can be inserted or deleted.
- Variables and register values can be examined or modified.
- Execution can be controlled (continue, step, next, stepi, etc).

VALGRIND MONITOR COMMANDS (COMPLETE LIST)

- (gdb) monitor v.info all_errors

  shows all errors found so far.

- (gdb) monitor v.info last_error

  shows the last error found.

- (gdb) monitor v.info n_errs_found

  shows the number of errors found so far (current value of the --vgdb-error).

- (gdb) monitor v.set {gdb_output | log_output | mixed_output}

  allows redirection of the Valgrind output (e.g. the errors detected by the tool).

- (gdb) monitor v.kill

  requests the gdbserver to kill the process.

BASICS MEMCHECK MONITOR COMMANDS (COMPLETE LIST)

- (gdb) monitor leak_check [full*| summary| xtleak]

  performs a leak check

- (gdb) monitor who_points_at <addr> [<len>]

  shows all the locations where a pointer to addr is found

DEMO 3: EX3.CPP

MEMCHECK

MEMORY ERROR DETECTOR

- It can detect common problems in C and C++ programs.
  - Accessing memory that you should not access
  - Using undefined values, i.e. values that have not been initialized
  - Incorrect freeing of heap memory
  - Memory leaks.

PROBLEMS LIKE THESE CAN BE DIFFICULT TO FIND BY OTHER MEANS, OFTEN REMAINING UNDETECTED FOR LONG PERIODS, THEN CAUSING OCCASIONAL, DIFFICULT-TO-DIAGNOSE CRASHES.

COMMON ERROR MESSAGES

- Illegal read / Illegal write errors
  - This happens when your program reads or writes memory at a place it shouldn’t
  - **Hint:** --read-var-info will run more slowly but may give a more detailed description of any illegal address
- Uninitialized values
  - An uninitialized-value use error is reported when your program uses a value that is undefined
  - **Hint:** --track-origins=yes easier to track down the root causes of uninitialized value errors

COMMON ERROR MESSAGES

- Memory leak detection

LEAK CASES

Pointer Chain	A Leak case	B Leak Case
R------------->B		Directly Reachable
R----->A---->B	Directly Reachable	Indirectly Reachable
R B		Directly Lost
R A---->B	Directly Lost	Indirectly Lost
R-------?----->B		Possibly Lost
R---->A--?-->B	Directly Reachable	Possibly Lost

LEAK KINDS RETURNED BY MEMCHECK

Still Reachable	Definitely Lost	Indirectly Lost	Possibly Lost

- **Hint:** --leak-check=full will give details for each definitely lost or possibly lost block, including where it was allocated

DEMO 4: EX4.CPP

MASSIF

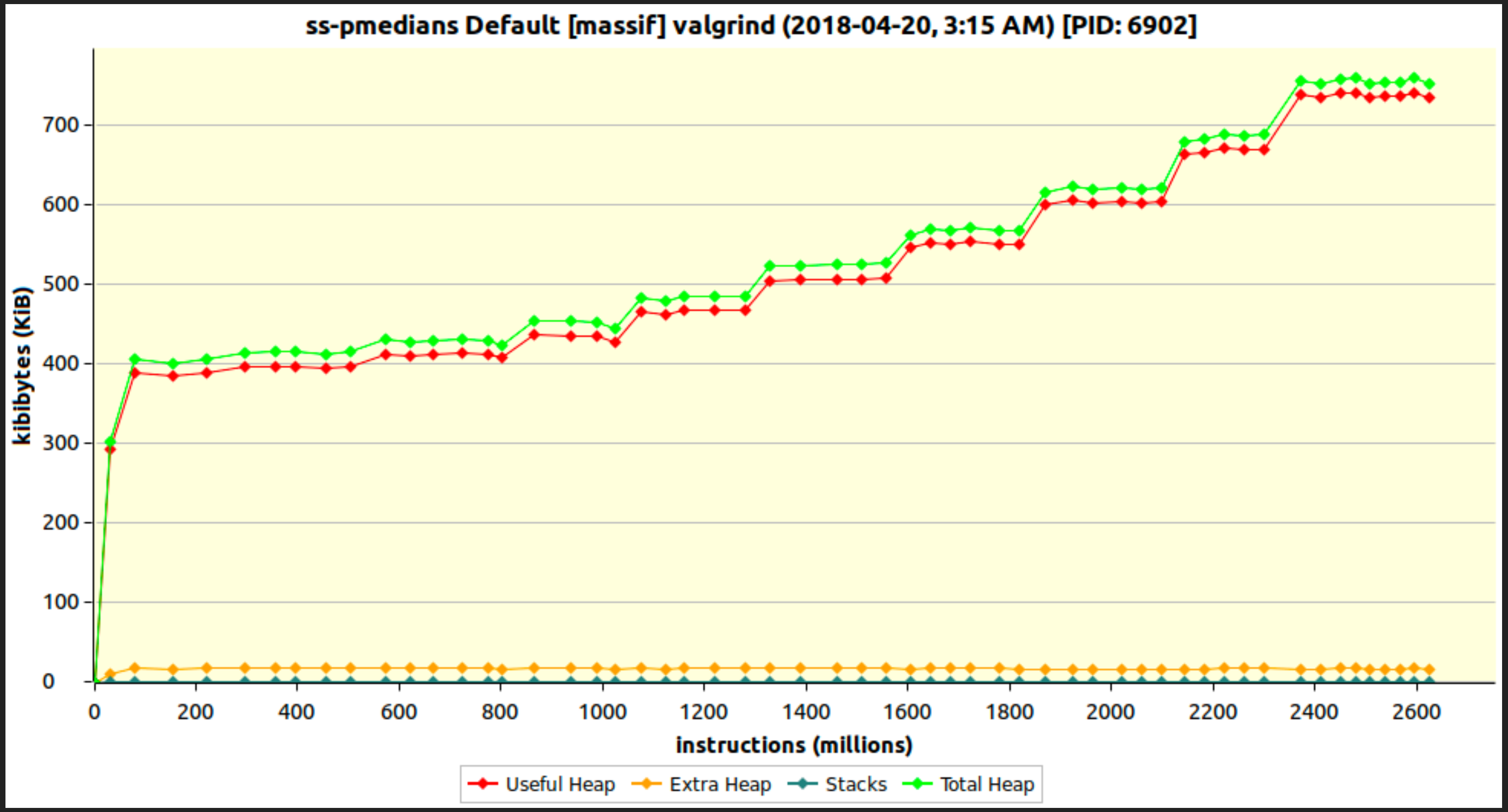
OVERVIEW

- Measures how much heap memory your program uses
- Can help you reduce the amount of memory your program uses.
    - It can speed up your program
    - Reduces the chance that it exhausts your machine's swap space.
- Detailed information that indicates which parts of your program are responsible for allocating the heap memory.

BASIC USAGE

```
▶ valgrind --tool=massif your-prog [your-prog-options]
```

- After program termination, a profile data file named massif.out.<pid> is generated
- Run ms_print to see the information gathered;
- (or use a visualization plugin such as eclipse)



DEMO 5 (USING ECLIPSE)

CALLGRIND

OVERVIEW

- Profiling tool that records the call history among functions in a program's run as a call-graph.
- Collects flat profile data: event counts (data reads, cache misses, etc.) are attributed directly to the function they occurred in.
- Allows you to find the specific call chains starting from ‘main’ in which the majority of the program's costs occur.

BASIC USAGE

```
valgrind --tool=callgrind your-prog [your-prog-options]
```

- After program termination, a profile data file named callgrind.out.<pid> is generated.
    - Use callgrind_annotate callgrind.out.<pid>
        - **Hint:** --inclusive=yes: Instead of using exclusive cost of functions as sorting order, use and show inclusive cost.
        - **Hint:** --auto=yes to get annotated source code for all relevant functions for which the source can be found
    - Graphical visualization of the data with KCachegrind

DEMO 6 (USING KCACHEGRIND)

CACHEGRIND

OVERVIEW

- Simulates how your program interacts with a machine's cache hierarchy
  - I1, D1 and LL (last-level) caches.
- Gather statistics of caches reads and misses
- These statistics are presented for the entire program and for each function in the program.
- Can also annotate each line of source code in the program with the counts that were caused directly by it.

BASIC USAGE

```
valgrind --tool=cachegrind your-prog [your-prog-options]
```

- After program termination, a profile data file named cachegrind.out.<pid> is generated.
  - Use cg_annotate cachegrind.out.<pid>
  - Use cg_diff first_file second_file to compare two profiling
- **Ideally used with optimizations flags**

DEMO 7 (USING ECLIPSE)

HELGRIND & DRD

THREAD ERROR DETECTORS:

- Helgrind (Eclipse support)
  - Detect synchronization errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.
    - Unlocking an invalid and not-locked mutex
    - Destroying an invalid or a locked mutex
    - Invalid or duplicate initialization of a pthread barrier
    - Waiting on an uninitialized pthread barrier
    - Many others

THREAD ERROR DETECTORS:

- DRD
  - Detecting errors in multithreaded C and C++ programs.
    - Data Race Detection
    - Lock contention. One thread blocks the progress of one or more other threads by holding a lock too long.
    - Deadlock. A deadlock occurs when two or more threads wait for each other indefinitely.
    - False sharing
    - Many others

CONCLUSIONS

WHY USE VALGRIND

- Valgrind will save you hours of debugging time
- Valgrind can help you speed up your programs
- Valgrind is free
- Valgrind is widely used
- Valgrind is actively maintained

WHEN SHOULD YOU USE VALGRIND?

- Easy answer: all the time.
    - After big changes
    - When a bug occurs (or is suspected)
    - Before a release
    - Whenever you want information about how your program is spending its time, or you want to speed it up

END