Valgrind Memory and profiling

START HERE

`valgrind ./<prog>`
Run under memcheck

`valgrind --leak-check=full ./<prog>`
Show each leak

`valgrind --track-origins=yes ./<prog>`
Find uninit sources

`valgrind --error-exitcode=1 ./<prog>`
Fail CI on errors

`valgrind --tool=callgrind ./<prog>`
Profile call costs

`valgrind --tool=massif ./<prog>`
Profile heap over time

`valgrind --tool=helgrind ./<prog>`
Find thread bugs

`valgrind --log-file=vg.%p.log ./<prog>`
Log to file

valgrind --tool=X synthetic CPU → runs → **Your program** 10-50x slower → reports → **Errors / out file** stderr, *.out.<pid> → read with → **Viewer** ms_print, *_annotate

CONCEPTS

Compile with -g and -O0 or -O1 so traces show file:line; skip -O2 inlining and sanitizers (ASan clashes with valgrind).

Memcheck is the default tool; --tool= picks another. Each tool is a separate run, they do not combine.

Reports show only what the run executed: a bug on an untested code path stays invisible.

RUNNING

`valgrind [opts] ./<prog> [args]`
Run program under valgrind

`valgrind --tool=<name> ./<prog>`
Pick memcheck, callgrind, massif

`valgrind -q ./<prog>`
Quiet: only print errors

`valgrind -s ./<prog>`
List errors and suppressions used

`valgrind --trace-children=yes ./<prog>`
Follow exec'd child processes
Use %p in file names so children do not overwrite each other.

`valgrind --num-callers=30 ./<prog>`
Deeper stack traces
Default is 12.

`valgrind --log-file=<file> ./<prog>`
Write messages to a file
%p expands to the PID.

`valgrind --error-exitcode=<n> ./<prog>`
Exit code n when errors found

`valgrind --read-var-info=yes ./<prog>`
Name variables in reports

`valgrind --vgdb=full ./<prog>`
Allow gdb attach, precise breaks

MEMCHECK LEAKS

`--leak-check=full`
List every leak with stack

MEMCHECK LEAKS (cont.)

`--show-leak-kinds=all`
Include reachable and indirect

`--show-leak-kinds=definite,possible`
Default set of kinds shown

`--errors-for-leak-kinds=<set>`
Which leaks count as errors

Leak kinds

Kind	Meaning
`definitely lost`	No pointer left: a real leak
`indirectly lost`	Only held by lost blocks
`possibly lost`	Only interior pointers remain
`still reachable`	Pointer exists at exit
`suppressed`	Matched a suppression

Fix definitely lost first; indirect ones vanish when their owner is freed. Still reachable is usually harmless.

MEMCHECK ERRORS

`--track-origins=yes`
Trace uninitialised values to source
Much slower, more memory; use only when you see an uninit error.

`--freelist-vol=<bytes>`
Bytes of freed blocks held back
Larger values catch use-after-free later; default 20000000.

Reading an error

Message	Usually means
Invalid read/write	Out of bounds or freed memory
Jump on uninitialised	Branch on unset variable
Invalid free()	Double free or bad pointer
Mismatched free	malloc/new/ new[] mix
Overlap in memcpy	Source and dest overlap
Syscall uninitialised	Unset bytes sent to kernel

The first error is often the cause of later ones; fix top-down and rerun. Read the 'Address ... is N bytes after a block' line.

SUPPRESSIONS

`--gen-suppressions=all`
Print a suppression per error

SUPPRESSIONS (cont.)

`--suppressions=<file>`
Load extra suppression file

Suppression entry

```
{
   libfoo_init_leak
   Memcheck:Leak
   match-leak-kinds: reachable
   fun:malloc
   ...
   obj:*/libfoo.so*
}
```

Suppress by narrow frames (fun:/obj:, with ... wildcards) so you hide a library's noise, not your own bugs.

CALLGRIND AND CACHEGRIND

`valgrind --tool=callgrind ./<prog>`
Record call graph and costs
Writes callgrind.out.<pid>.

`callgrind_annotate callgrind.out.<pid>`
Text report by function

`callgrind_annotate --inclusive=yes <file>`
Sort by inclusive cost

`callgrind_annotate --tree=caller <file>`
Show callers per function

`kcachegrind callgrind.out.<pid>`
Browse graph in a GUI

`--toggle-collect=<function>`
Collect only inside a function

`--cache-sim=yes`
Add cache miss counters

`callgrind_control -d`
Dump profile of a running program

`valgrind --tool=cachegrind ./<prog>`
Cache and branch simulation
Writes cachegrind.out.<pid>; add --branch-sim=yes for branches.

`cg_annotate cachegrind.out.<pid>`
Report cachegrind results

`cg_annotate --diff <file1> <file2>`
Compare two cachegrind runs

Costs are instruction counts, not seconds: fully reproducible, but use perf for real timing.

MASSIF AND DHAT

`valgrind --tool=massif ./<prog>`
Heap usage over time
Writes massif.out.<pid>.

`ms_print massif.out.<pid>`
Graph and snapshot tree

`--time-unit=B`
Time in bytes allocated, reproducible

`--pages-as-heap=yes`
Profile all mapped pages, not malloc

`valgrind --tool=dhat ./<prog>`
Per-site allocation hot spots
Shows lifetimes, usage and short-lived allocations; view the JSON in the dh_view.html bundled with valgrind.

Peak heap is marked in the ms_print graph with a # column; the detailed snapshot there shows who allocated it.

THREAD CHECKERS

`valgrind --tool=helgrind ./<prog>`
Find races, lock misuse, deadlocks

`valgrind --tool=drd ./<prog>`
Alternative thread checker, less memory

Use pthreads or C++ std::thread with libpthread; hand-rolled atomics and futexes show up as false positives. Annotate or suppress them.

RECIPES

CI memory gate

```
valgrind --leak-check=full --
show-leak-kinds=definite \
  --errors-for-leak-
kinds=definite --error-
exitcode=1 \
  --log-file=vg.%p.log ./<prog>
<args>
```

Hunt an uninitialised value

```
gcc -g -O0 -o <prog> <prog>.c
valgrind --track-origins=yes --
num-callers=30 ./<prog>
```