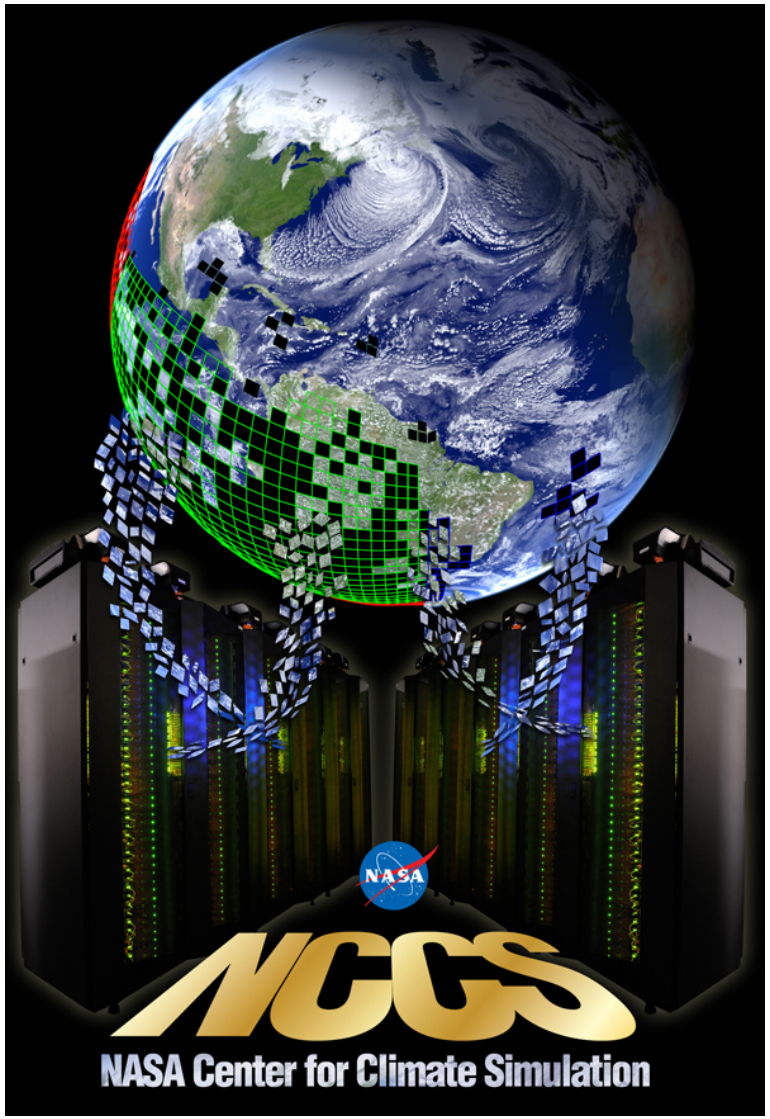
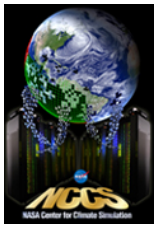
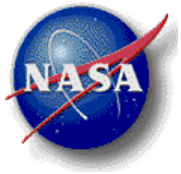


NCCS Brown Bag Series

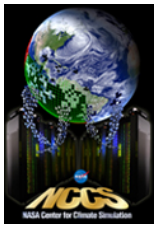


Using Valgrind to Detect Memory Leaks

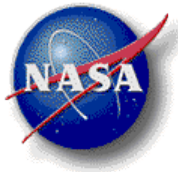
Chongxun (Doris) Pan
doris.pan@nasa.gov
November 8, 2012



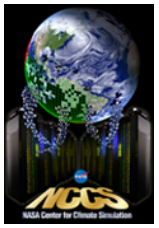
Agenda



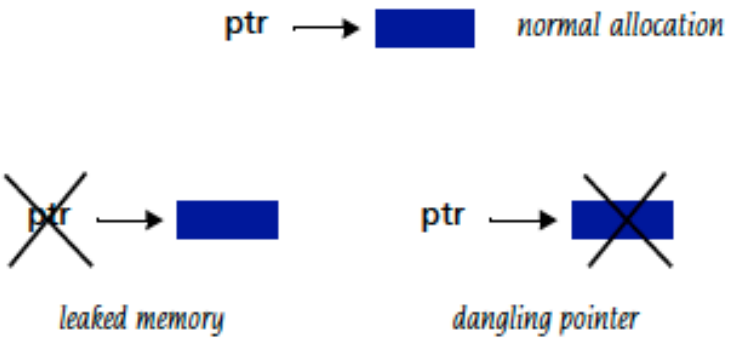
- What is a Memory Leak?
- What are the available tools to detect memory errors for my applications?
- What is Valgrind?
- How does it work? How do I use it on Discover?



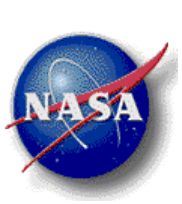
What is a Memory Leak?



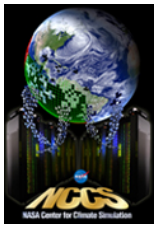
- A memory allocation that does not have a corresponding de-allocation



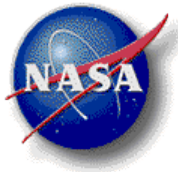
- For a given scale or platform or problem, they may not be fatal
- Failures could occur until modification, reuse of a component, or moving the application to a different cluster with a new OS



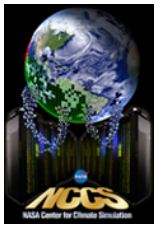
Detection Tools for Memory Leaks/Errors



- A number of tools available to track memory usage for C/C++ using wrapper libraries for malloc/free (for C) or new/delete (for C++)
  - LeakTracer, ccmalloc, Cmemleak, NJAMD, mpatrol…
- Only few tools available for Fortran programmers
  - Valgrind/Memcheck
  - TotalView/MemScape
    http://www.nccs.nasa.gov/images/Totalview-Part2-Doris.pdf
  - TAU http://www.nccs.nasa.gov/images/TAU-brownbag.pdf
  - Intel Inspector XE (Part of Parallel Studio XE, not yet installed on Discover. Supporting Intel compiler 12+)
    http://software.intel.com/en-us/intel-inspector-xe

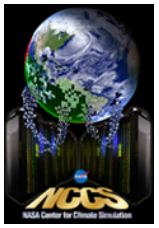


What is Valgrind?



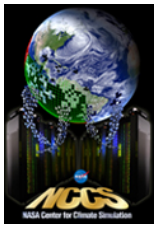
- Valgrind is a suite of command line tools for both debugging and profiling codes on Linux, including
  - Memcheck -- A memory error checking tool
    - Valgrind's most popular tool. Often synonymous with "Valgrind"
  - Cachegrind – A cache simulator
  - Callgrind – Extension of Cachegrind. A call-graph profiler
  - Massif -- A heap profiler
- This talk focuses on Memcheck. Other tools may not necessarily be what you need, but demonstrate things that you could do with Valgrind.

What is Valgrind?

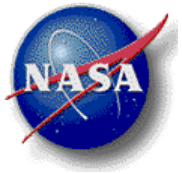


- Largely aimed at C/C++. But it can be used on programs written partly or entirely in Fortran, Java, Perl, Python, assembly code, etc.
- Can be used with existing executables without recompiling or relinking. But the –g –O0 (for Intel compilers, -g implies –O0) flags are recommended because the output will be more useful, including the line number of the source code.

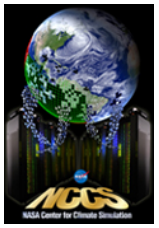
What Errors does Valgrind/Memcheck Detect?



- Reading/writing freed memory or incorrect memory areas
- Uninitialized values
- Incorrect freeing of memory, such as double freeing heap blocks
- Misuse of functions for memory allocations: new(), malloc(), free(), deallocate(), etc.
- Memory leaks - unintentional memory consumption often related to program logic flaws which lead to loss of memory pointers prior to deallocation

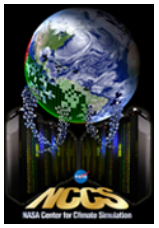


Limitations of Valgrind



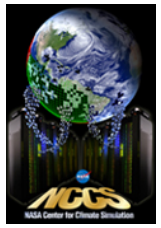
- Does not perform bounds checking on static arrays (i.e., memory allocated on the stack)
- Only checks programs dynamically -- May report no errors on a particular input set although the program contains bugs
- Consumes more memory (~2x)
- Slows down the programs (10x and more)
- Optimized binaries can cause Valgrind to wrongly report uninitialized value errors

Limitations of Valgrind (*Cont'd*)



- You will encounter a lot of false positives, specially for Fortran IO routines. See later slides on how to filter those out.
- Limited support for debugging parallel programs
  - Helgrind: debugging programs with POSIX pthreads threading primitives. No OpenMP support.
  - MPI support consists of a library of wrapper functions for PMPI_* interface, buildable with mpicc
  - Expect a lot of false errors!
- NOT suitable to debug large HPC applications

Versions of Valgrind Installed on Discover



- The default version after the SP1 upgrade is 3.5.0
- The latest version is built on SLES11/SP1 under

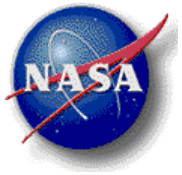
/discover/nobackup/cpan2/lib/valgrind-3.8.1/build-SP1

```
discover15:$ /usr/bin/valgrind --version
valgrind-3.5.0
discover15:$ which valgrind
/discover/nobackup/cpan2/lib/valgrind-3.8.1/build-SP1/bin/valgrind
discover15:$ valgrind –version
valgrind-3.8.1
discover15:$ valgrind --help
```

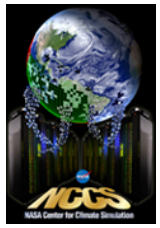
valgrind [valgrind-options] ./prog.x [prog-options]

"--tool=memcheck –leak-check=summary" is the default

--log-file=filename can direct output to a file



A few simple examples – Ex 1: Reading/writing out-of-bound



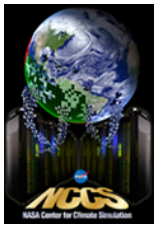
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int i;
  int *a = malloc(sizeof(int) * 10);
  if (!a) return -1;
  for (i = 0; i < 11; i++){
    a[i] = i; /* problem here */
  }
  free(a);
  return 0;
}
```

All the example codes presented are located on

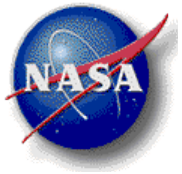
/discover/nobackup/cpan2/Valgrind

```
discover15:$ module list
Currently Loaded Modulefiles:
  1) comp/intel-12.1.0.233      3) tool/tview-8.9.2.2
  2) mpi/impi-4.0.1.007-beta    4) other/comp/gcc-4.6.3-sp1
discover15:$ icc -g -O0 -o ex1 ex1.c (or using gcc)
discover15:$ valgrind ./ex1
==1896== Memcheck, a memory error detector
==1896== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==1896== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==1896== Command: ./ex1
==1896==
==1896== Invalid write of size 4
==1896==    at 0x4005BC: main (ex1.c:9)
==1896==  Address 0x57fc068 is 0 bytes after a block of size 40 alloc'd
==1896==    at 0x4C2756F: malloc (vg_replace_malloc.c:270)
==1896==    by 0x40057F: main (ex1.c:6)
==1896==
==1896==
==1896== HEAP SUMMARY:
==1896==     in use at exit: 0 bytes in 0 blocks
==1896==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==1896==
==1896== All heap blocks were freed -- no leaks are possible
==1896==
==1896== For counts of detected and suppressed errors, rerun with: -v
==1896== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 5 from
5)
```

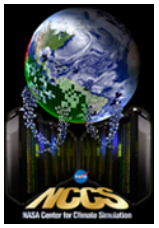
What we learned from Example 1:



- You can ignore “1896”, the process ID
- The first line (“Invalid write…”) tells the type of the error, followed by a stack trace showing where the problem occurred. If the stack trace is not big enough, use –num-caller=<*number*> option
- Notice that some errors are suppressed -- this is because they could be from standard library routines rather than your own code.

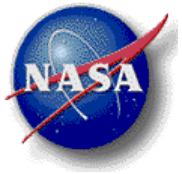


Ex 2: Uninitialized values

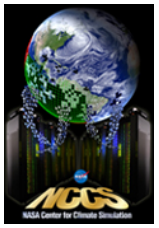


```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv){
  int i;
  int a[10];
  for (i = 0; i < 9; i++)
    a[i] = i;

  for (i = 0; i < 10; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
  return 0;
}
```

```
discover15:$ icc -g -O0 -o ex2 ex2.c
discover15:$ ./ex2
0 1 2 3 4 5 6 7 8 4195792
discover15:$ valgrind ./ex2
…
==5871== Use of uninitialised value of size 8
==5871==    at 0x52DCA43: _itoa_word (in /lib64/libc-2.11.1.so)
==5871==    by 0x52DFAD6: vfprintf (in /lib64/libc-2.11.1.so)
==5871==    by 0x52E7AA9: printf (in /lib64/libc-2.11.1.so)
==5871==    by 0x40058A: main (ex2.c:11)
==5871==
==5871== Conditional jump or move depends on uninitialised value(s)
==5871==    at 0x52DCA4D: _itoa_word (in /lib64/libc-2.11.1.so)
==5871==    by 0x52DFAD6: vfprintf (in /lib64/libc-2.11.1.so)
==5871==    by 0x52E7AA9: printf (in /lib64/libc-2.11.1.so)
==5871==    by 0x40058A: main (ex2.c:11)
…
0 1 2 3 4 5 6 7 8 4195792
==5871==
==5871== HEAP SUMMARY:
==5871==     in use at exit: 0 bytes in 0 blocks
==5871==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==5871==
==5871== All heap blocks were freed -- no leaks are possible
==5871==
==5871== For counts of detected and suppressed errors, rerun with: -v
==5871== Use --track-origins=yes to see where uninitialised values come from
==5871== ERROR SUMMARY: 17 errors from 5 contexts (suppressed: 5 from 5)
```



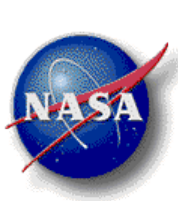
What we learned from Example 2:



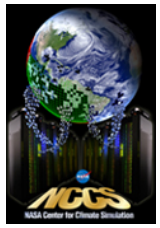
- If you run with the option --track-origins=yes, valgrind will give additional information about where the uninitialized values come from.

```
==29315== Conditional jump or move depends on uninitialised value(s)
==29315==    at 0x52E029B: vfprintf (in /lib64/libc-2.11.1.so)
==29315==    by 0x52E7AA9: printf (in /lib64/libc-2.11.1.so)
==29315==    by 0x40058A: main (ex2.c:11)
==29315==  Uninitialised value was created by a stack allocation
==29315==    at 0x400514: main (ex2.c:4)
```

- Notice that the output of the program and the output of valgrind are interleaved. To redirect the output to a separate file, using --log-file=filename

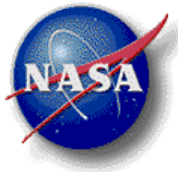


Ex 3: Memory leaks

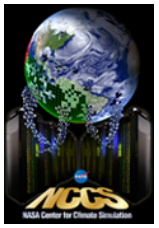


```
program ex3
integer*4, parameter :: array_mb = 500
integer*4 :: i, im, is
integer*4, pointer, dimension(:) :: p_array
integer*4 :: mb = 1024*1024/4

im = array_mb * mb

do i = 1,2
! explicit deallocation P_array would fix
this problem
allocate (p_array(im), stat=is)
call use_array (p_array, im)
write (*,*) i,' p_array allocated'
end do

end

subroutine use_array (array, im)
integer*4 im, array(im), i

do i = 1,im
array(i) = im-i
end do
end
```

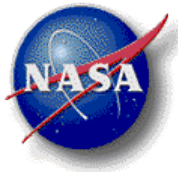
```
discover15:$ ifort -g -O0 -o ex3 ex3.f90
discover15:$ ./ex3
Exit Normally
discover15:$ valgrind --suppressions=./myvalgrind.supp --leak-
check=full ./ex3
…
==3548== HEAP SUMMARY:
==3548==     in use at exit: 1,048,576,032 bytes in 3 blocks
==3548==   total heap usage: 10 allocs, 7 frees, 1,048,588,551 bytes allocated
==3548==
==3548== 524,288,000 bytes in 1 blocks are possibly lost in loss record 3 of 3
==3548==    at 0x4C2756F: malloc (vg_replace_malloc.c:270)
==3548==    by 0x406653: for_allocate (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==3548==    by 0x402B94: MAIN__ (ex3.f90:11)
==3548==    by 0x402AAB: main (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==3548==
==3548== LEAK SUMMARY:
==3548==    definitely lost: 0 bytes in 0 blocks
==3548==    indirectly lost: 0 bytes in 0 blocks
==3548==      possibly lost: 524,288,000 bytes in 1 blocks
==3548==    still reachable: 524,288,032 bytes in 2 blocks
==3548==         suppressed: 0 bytes in 0 blocks
==3548== Reachable blocks (those to which a pointer was found) are not
shown.
==3548== To see them, rerun with: --leak-check=full --show-reachable=yes
==3548==
==3548== For counts of detected and suppressed errors, rerun with: -v
==3548== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 40 from
37)
```



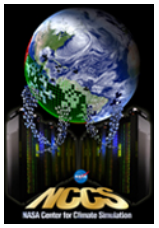
What we learned from Example 3:



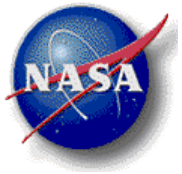
- Several kinds of leaks reported:
  - "definitely lost": leaking memory -- fix it!
  - “possibly lost”: general indicates leaking memory – fix it!
  - “indirect lost”: usually disappear if the “definitely” lost block that caused the indirect leak is fixed.
- Recommend to always use --leak-check=full for leak detection. It will give details for each definitely lost or possibly lost block.
- To find absolutely every unpaired call to allocate/ deallocate, you'll need to use the --show-reachable=yes option.



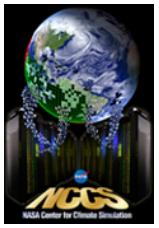
Error Suppressions



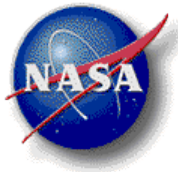
- Valgrind detects many errors (some are false positives) in system C or Fortran libraries.
- At startup it reads a default suppression file $PREFIX/lib/config/default.supp
- You can create your own suppression file(s) -- very useful to suppress errors that you know are false positives.
- Approach: Use --gen-suppressions=all|yes option to generate suppressions, create your own suppression file, and apply them using --suppressions=/path/to/myfile.supp



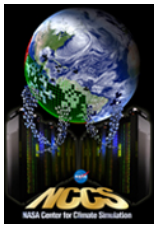
Error Suppressions



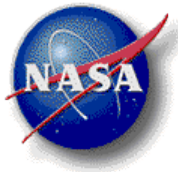
```
discover15:$ valgrind --gen-suppressions=all ./ex3
…
==8131== Use of uninitialised value of size 8
==8131==    at 0x429EA1: for__add_to_lf_table (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==8131==    by 0x441B7D: for__open_proc (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==8131==    by 0x42FF5A: for__open_default (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==8131==    by 0x409019: for_write_seq_lis (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==8131==    by 0x402EF2: MAIN__ (ex3.f90:13)
==8131==    by 0x402AAB: main (in /gpfsm/dnb31/cpan2/Valgrind/ex3)
==8131==
{
   <insert_a_suppression_name_here>
   Memcheck:Value8
   fun:for__add_to_lf_table
   fun:for__open_proc
   fun:for__open_default
   fun:for_write_seq_lis
   fun:MAIN__
   fun:main
}
…
discover15:$ vim myvalgrind.supp
discover15:$ valgrind --suppressions=./myvalgrind.supp ./ex3
```



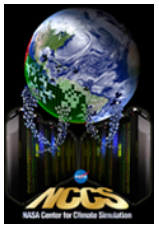
Debugging MPI Programs



- You can always do “mpirun –np *n* valgrind ./exe ..” but expect a LOT of false positives that Memcheck reports for MPI calls
- Valgrind supports a library of wrapper functions for the PMPI_* interface, buildable with mpicc only
- The wrappers incorporate into the application’s memory space, either by direct linking or by LD_PRELOAD, reducing the number of false errors on MPI applications

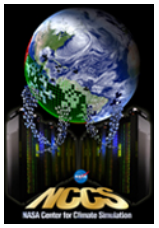


Sample PBS Script

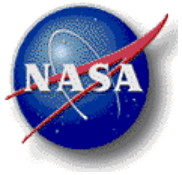


```
#!/usr/bin/csh
#PBS -N Test_Valgrind
#PBS -l walltime=1:00:00
#PBS –l select=2:ncpus=12:mpiprocs=12
#PBS -j oe
#PBS -o PBS_output
#PBS -W umask=022

module purge
module load other/comp/gcc-4.6.3-sp1 other/mpi/openmpi/1.6.3-gcc-4.6.3

cd /discover/nobackup/cpan2/Valgrind
mpif90 –g –O0 –o testmpi testmpi.f90

setenv LD_PRELOAD /discover/nobackup/cpan2/lib/valgrind-3.8.1/build-SP1-mpi/lib/valgrind/
libmpiwrap-amd64-linux.so
setenv MPIWRAP_DEBUG quiet

mpirun –np 24 valgrind --log-file=out.%p ./testmpi
```

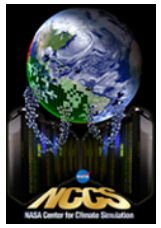
Notes on Debugging MPI Programs



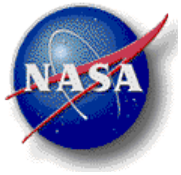
- Compile your application with the **same** compiler and mpi module that we built the wrappers with. Using a different MPI-library will generate **a lot more** false messages in your output file.
- %p is replaced with the current process ID. --log-file=out.%p is very useful for programs that invoke multiple processes.
- The wrapping is done at the MPI interface, so there still could be a large number of false errors reported in the MPI implementation below the interface.
- But you know how to suppress them now!



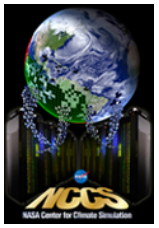
Useful Options for Valgrind (version 3.8.1)



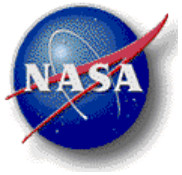
--help or -h	Print help command
--help-debug	Print help command plus debugging option
--quiet or -q	Show only the error message
--version	Show version
--log-file=<file>	Log Valgrind output messages to <file>
--num-callers=<number> [default:12]	Show <number> callers in stack traces
--gen-suppressions=no|yes|all [default: no]	print suppressions for errors
--suppressions=<filename>	Use the file described in <filename> to suppress errors



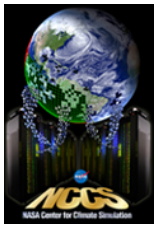
Useful Options for Memcheck



--leak-check=no|summary|full [default: summary]	Valgrind tracks all memory block allocations. When the program finishes it prints which blocks have not been freed. The option full shows a lot of detail.
--show--reachable=no|yes [default: no]	Print some information about blocks of memory not deallocated but which have references.
–leak-resolution=low|med|high [default: high]	If the option low is enabled each single message will print only the first time it will be matched in leak stack traces. High prints the same message for each occurrence.
--track-origins=no|yes [default: no]	Show origins of undefined values or not

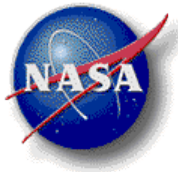


Setting Default Options

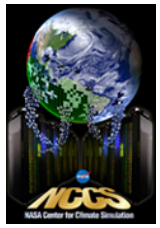


- Valgrind also reads options from three places, in the listed order of precedence
  - $HOME/.valgrindrc
  - The env variable $VALGRIND_OPTS
  - ./.valgrindrc
- Any tool-specific options in $VALGRIND_OPTS or the .valgrindrc files should be prefixed with the tool name and a colon, e.g.,

```
discover15:$ cat ~/.valgrindrc
--memcheck:leak-check=full
```



More info and references…



- Find further information on the Valgrind homepage

http://www.valgrind.org

- This presentation, as well as other NCCS brownbag talks, are located at

http://www.nccs.nasa.gov/list_brown_bags.html