Short Valgrind Tutorial

Patrick Sanan
patrick.sanan@erdw.ethz.ch

March 6, 2017

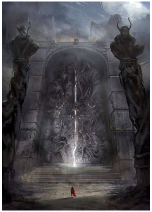
This presentation (with examples):
https://bitbucket.org/psanan/valgrind_tutorial

http://valgrind.org/images/st-george-dragon.png

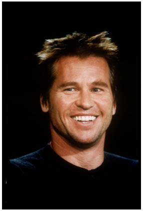
Introduction

What's in a name?

- Valgrind, the gate to Valhalla



- Pronounced like "Val grinned"



- http://valgrind.org/docs/manual/faq.html#faq.pronounce

What's the Problem?

- In programming languages like C and Fortran, you are responsible for your own dynamic memory management
- You reserve memory on the *heap* with which to do your computations
- You return this memory when you are finished with it
- You are responsible for only accessing memory locations that you have reserved
- You can make mistakes!
  - Forgetting to return memory
  - Reading and writing into memory you haven't reserved
  - Using uninitialized memory to control logic
- These are all **unacceptably bad things to do**, but the compiler can't warn you*
- Valgrind helps detect and locate these mistakes, using just your executable

Installation (aka the worst part)

Obtaining Valgrind

- Bad news: Valgrind does **not** work on OS X 10.11 and 10.12 (it worked sporadically before that).
  - MacPorts will install it for you, but it doesn't work!
- Valgrind works very well on most Linux systems, and is often available through a package manager

```
sudo apt-get install valgrind
```

- Valgrind is actually quite easy to download and build yourself (but probably still won't work on recent OS X systems)
  http://valgrind.org/downloads/current.html

Building Valgrind 3.12.0 on Euler (thanks to Ilya Fomin)

- Get the source onto Euler

```
ssh euler
wget http://valgrind.org/downloads/valgrind-3.12.0.tar.bz2
tar xvf valgrind-3.12.0.tar.bz2
```

- configure, build, and install. It will take a long time. Don't move the binary from this location!

```
cd valgrind-3.12.0
mkdir -p $HOME/valgrind_install
./configure --prefix=$HOME/valgrind_install
make && make install
```

- Add to your path

```
export PATH=$PATH:$HOME/valgrind_install  # can go in your login file (e.g. .
    bashrc)
which valgrind
```

- Make sure it runs (by testing the built in `ls -l` function)

```
valgrind ls -l
echo "valgrind ls -l" > tmp.sh && bsub < tmp.sh && rm tmp.sh
  # examine resulting lsf.xxx
```

Basic Usage

Using Valgrind

- Valgrind is very easy to use; just supply your program, with arguments

```
valgrind ./my_program -arg1 -arg2
valgrind -- ./my_program -arg1 -arg2 # sometimes required
```

- It helps greatly to include debugging symbols (compile with -g[1])
- Output can be more meaningful if you compile without optimization, e.g. `-O0`.
- You can redirect valgrind's output to a file with `--log-file`, for example

```
valgrind --log-file=valgrind.log ./my_program
```

- The combined program output and valgrinds output can be directed both the screen and to a file with standard UNIX tools:

```
valgrind ./my_program 2>&1 | tee valgrind_all.txt
```

[1] unless you care about executable size, you should also do this with optimized builds

Hello, World

- By default, valgrind uses the Memcheck tool, which checks for dynamic memory errors.
- See `examples/1_hello` (you need working gcc and/or gfortran compilers)
- I've provided most of the examples in C and fortran, but will mostly refer to the C versions here

```
cd examples/1_hello_world/c
make
./hello
valgrind ./hello
```

```
cd examples/1_hello_world/Fortran
make
./hello
valgrind ./hello
```

- Examine the output (on the next slide)

Valgrind Output

```
$ valgrind ./hello
==16349== Memcheck, a memory error detector
==16349== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16349== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==16349== Command: ./hello
==16349==
Hello, World!
==16349==
==16349== HEAP SUMMARY:
==16349==     in use at exit: 0 bytes in 0 blocks
==16349==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==16349==
==16349== All heap blocks were freed -- no leaks are possible
==16349==
==16349== For counts of detected and suppressed errors, rerun with: -v
==16349== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- There were no warning messages, and no leaks are reported, which means that valgrind detected no problems!
- This is commonly talked about as being "valgrind clean"

Dynamic memory errors: Memcheck

- Memcheck is the default tool, so often when people say "valgrind," this is what they mean
- Memcheck tries to detect and warn about errors when using dynamic memory (on the heap)
- It works by running your code on a virtual machine, keeping track of every single bit of memory by attaching a second, "is valid" bit. Thus, you would expect it to at least double the amount of required memory.

Memory Leaks (forgotten frees)

- In C (and C++) and Fortran, you can dynamically allocate memory
- This means requesting a chunk of memory from the operating system
- It's up to the programmer to return the memory when finished, so that it can be used elsewhere
- Failing to do this causes **insidious bugs**. There is no effect on the performance on the program .. until no more memory is available and the program crashes
- If a forgotten free occurs inside a timestepping loop, the program will increase its memory usage without bound, given enough time (bad news for modellers)

Forgotten Free Example

- See `examples/2_forgotten_free/c` and `make`

```
$ valgrind ./forgotten_free
==16990== Memcheck, a memory error detector
==16990== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16990== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==16990== Command: ./forgotten_free
==16990==
==16990==
==16990== HEAP SUMMARY:
==16990==     in use at exit: 40 bytes in 1 blocks
==16990==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==16990==
==16990== LEAK SUMMARY:
==16990==    definitely lost: 40 bytes in 1 blocks
==16990==    indirectly lost: 0 bytes in 0 blocks
==16990==      possibly lost: 0 bytes in 0 blocks
==16990==    still reachable: 0 bytes in 0 blocks
==16990==         suppressed: 0 bytes in 0 blocks
==16990== Rerun with --leak-check=full to see details of leaked memory
==16990==
==16990== For counts of detected and suppressed errors, rerun with: -v
==16990== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- The important line here is this one, but where's the leak?

```
==16990==    definitely lost: 40 bytes in 1 blocks
```

```
$ valgrind --leak-check=full ./forgotten_free
==17011== Memcheck, a memory error detector
==17011== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17011== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==17011== Command: ./forgotten_free
==17011==
==17011==
==17011== HEAP SUMMARY:
==17011==     in use at exit: 40 bytes in 1 blocks
==17011==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==17011==
==17011== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==17011==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
    )
==17011==    by 0x40053E: main (forgotten_free.c:4)
==17011==
==17011== LEAK SUMMARY:
==17011==    definitely lost: 40 bytes in 1 blocks
==17011==    indirectly lost: 0 bytes in 0 blocks
==17011==      possibly lost: 0 bytes in 0 blocks
==17011==    still reachable: 0 bytes in 0 blocks
==17011==         suppressed: 0 bytes in 0 blocks
==17011==
==17011== For counts of detected and suppressed errors, rerun with: -v
==17011== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

We can fix our code by adding the missing `free(a)`

Forgotten Frees in Fortran

- You can repeat the above in `examples/1_forgotten_free/fortran`
- Note that allocatable arrays are deallocated for you when they go out of scope!
- Using pointers behaves similarly to C: see `forgotten_free_2.c`.

Invalid Reads

- C and Fortran let the programmer interact with memory directly by address.
- This is efficient, but allows the programer to read memory locations which they have not allocated.
- This is almost always an error, because nothing can be assumed about the values of these memory locations.
- For modellers, this is dangerous: the behavior can be *non-deterministic*, but this fact will often not manifest until one changes environments (say move from debugging on a laptop to running on the cluster)

Invalid Read Example

- examples/2_invalid_read/c

```
$ valgrind ./invalid_read
==17965== Memcheck, a memory error detector
==17965== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17965== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==17965== Command: ./invalid_read
==17965==
==17965== Invalid read of size 4
==17965==    at 0x4005D7: main (invalid_read.c:6)
==17965==  Address 0x51ff068 is 0 bytes after a block of size 40 alloc'd
==17965==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
     linux.so)
==17965==    by 0x4005CE: main (invalid_read.c:5)
==17965==
b = 0
==17965==
==17965== HEAP SUMMARY:
==17965==     in use at exit: 0 bytes in 0 blocks
==17965==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==17965==
==17965== All heap blocks were freed -- no leaks are possible
==17965==
==17965== For counts of detected and suppressed errors, rerun with: -v
==17965== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Invalid Writes

- The programmer is also free to write to memory locations that they haven't reserved for themselves
- This can lead to some **very** confusing bugs
  - Sometimes nothing happens, because your program doesn't ever use the value you wrote
  - You can write to a location used for something else, in which case an error may be observed in a completely-unrelated part of the code
  - The details of memory allocation are handled by the OS, so the effect is very non-deterministic
- This is also unacceptable for modellers, because data can be corrupted

Invalid Write Example

- `examples/4_invalid_write/c`
- This example fills two arrays with values 1 to 10, and prints them
- There is a mistake in one of the loop bounds
- For me, this causes one of the arrays to have the wrong values, and the OS actually reports an error, but neither of these is guaranteed to happen!

```
$ ./invalid_write
a: 0 1 2 3 4 5 6 7 8 9
b: 12 13 14 15 16 17 18 19 20 21
*** Error in './invalid_write': free(): invalid next size (fast): 0
     x00000000006c5010 ***
Aborted (core dumped)
```

- Valgrind can pinpoint the error

```
$ valgrind ./invalid_write
==18370== Memcheck, a memory error detector
==18370== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==18370== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==18370== Command: ./invalid_write
==18370==
==18370== Invalid write of size 4
==18370==    at 0x40067D: main (invalid_write.c:10)
==18370==  Address 0x51ff068 is 0 bytes after a block of size 40 alloc'd
==18370==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
     )
==18370==    by 0x40061E: main (invalid_write.c:6)
==18370==
a: 0 1 2 3 4 5 6 7 8 9
b: 28 29 30 31 32 33 34 35 36 37
==18370==
==18370== HEAP SUMMARY:
==18370==     in use at exit: 0 bytes in 0 blocks
==18370==   total heap usage: 2 allocs, 2 frees, 80 bytes allocated
==18370==
==18370== All heap blocks were freed -- no leaks are possible
==18370==
==18370== For counts of detected and suppressed errors, rerun with: -v
==18370== ERROR SUMMARY: 80 errors from 1 contexts (suppressed: 0 from 0)
```

Uninitialized Values

- When you receive memory from the OS, the values are not initialized
- Thus, basing program logic on these values will not behave deterministically
- However, in many cases, these values will in fact be zero or some other constant value [2]
- This can cause bugs which are very hard to notice in standard ways, because they often only manifest when moving to a new system or compiler (and for modellers, anything which changes between debugging machine and cluster is bad news)
- If you want values to be zero, set them to zero (or use `calloc` in C)
- It is not an error to manipulate uninitialized values, just to base decisions on them; for that reason, valgrind will not report something like this as an error

```
float *a = (float*) malloc(10*sizeof(float));
a[7] += 1.3;
```

[2] perhaps you have noticed how certain exponents like e-310 indicate an uninitialized floating point value

Uninitialized Values Example

- `examples/5_uninitialized_value`
- This example bases an if statment on an uninitialized value
- This value is zero, but I cannot assume that to always be true
- Valgrind will pinpoint the error, but not the precise value

```
$ valgrind ./uninitialized_value
==19492== Memcheck, a memory error detector
==19492== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==19492== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==19492== Command: ./uninitialized_value
==19492==
==19492== Conditional jump or move depends on uninitialised value(s)
==19492==    at 0x400617: main (uninitialized_value.c:9)
==19492==
a[7]  >= 0
==19492==
==19492== HEAP SUMMARY:
==19492==     in use at exit: 0 bytes in 0 blocks
==19492==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==19492==
==19492== All heap blocks were freed -- no leaks are possible
==19492==
==19492== For counts of detected and suppressed errors, rerun with: -v
==19492== Use --track-origins=yes to see where uninitialised values come from
==19492== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Tracking Origins of Uninitialized Values

- Valgrind will also tell you where the uninitialized value was allocated, if you ask (it won't by default because this is slower)

```
$ valgrind --track-origins=yes ./uninitialized_value
==19513== Memcheck, a memory error detector
==19513== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==19513== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==19513== Command: ./uninitialized_value
==19513==
==19513== Conditional jump or move depends on uninitialised value(s)
==19513==    at 0x400617: main (uninitialized_value.c:9)
==19513==  Uninitialised value was created by a heap allocation
==19513==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
     )
==19513==    by 0x4005CE: main (uninitialized_value.c:5)
==19513==
a[7]  >= 0
==19513==
==19513== HEAP SUMMARY:
==19513==     in use at exit: 0 bytes in 0 blocks
==19513==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==19513==
==19513== All heap blocks were freed -- no leaks are possible
==19513==
==19513== For counts of detected and suppressed errors, rerun with: -v
==19513== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind and MPI

- Valgrind can be run on each rank in an MPI application

```
mpiexec -np 4 valgrind ./my_parallel_app -arg   # Yes
```

- It's important to get the order of the arguments correct. This is probably not what you want:

```
valgrind mpiexec -np 4 ./my_parallel_app -arg   # NO
```

- Most MPI implementations will produce many valgrind warnings
- A practical way to get an MPI installation that doesn't do this is to have PETSc download and install MPICH [3]

[3] `./configure --download-mpich` and look in `PETSC_ARCH/bin/` for `mpicc`,`mpiexec`, etc.

- See `examples/6_mpi/c`
- This is the first example of what valgrind often looks like "in the wild", when you have to learn to ignore messages from code that you aren't responsible for
- First, try this:

```
valgrind mpiexec -np 2  ./reduction
```

- In my case, this does not reveal anything about my application - it's telling me abouty the `mpiexec` program!
- Instead, try the following, which reveals the logical error in the code, amongst many other warnings

```
$  mpiexec -np 2 valgrind  ./reduction
```

More on the Leak Summary

Which of these should I worry about?

```
LEAK SUMMARY:
==19754==    definitely lost: 51,172 bytes in 70 blocks
==19754==    indirectly lost: 14,378 bytes in 39 blocks
==19754==      possibly lost: 0 bytes in 0 blocks
==19754==    still reachable: 127,364 bytes in 528 blocks
==19754==         suppressed: 0 bytes in 0 blocks
==19754== Rerun with --leak-check=full to see details of leaked memory
```

Definitely Lost

- These indicate blocks of memory to which no pointer exists.
- Unless you are forced to use library code (such as MPI) which you can't fix..
- **Fix these!**

Indirectly Lost

- These are blocks of memory for which a pointer exists, but that pointer is in lost memory
- These are just as bad as direct losses, since the memory can't be freed, so if it's your code ..
- **Fix these!**

Possibly Lost

- These are cases where a pointer to the block doesn't exist, but it might still be possible to free the memory by manipulating a existing pointer to the middle of the block.
- Unless you are performing complicated pointer operations and know why this might be okay, if they occur in your code ..
- **Fix these!**

Still Reachable

- These are blocks which, at the end of the program, are not freed, though pointers to them exist.
- This is mostly harmless (the OS frees everything for you), so..
- **Don't worry about these**
- You may notice that valgrind will often report fewer frees than allocations, and this is one reason.

Static Memory Errors

Static Memory Errors

- **Memcheck does NOT detect illegal use of static (stack) arrays, even though these can cause all the same sorts of bugs!**

```
int a[3];
a[10] = 1; /* fine according to memcheck */
```

- See `examples_7_static_error`.
- Note that valgrind does not catch the errors here

```
$ valgrind ./static_error
==20437== Memcheck, a memory error detector
==20437== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==20437== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==20437== Command: ./static_error
==20437==
b = 0
c = 3
==20437==
==20437== HEAP SUMMARY:
==20437==     in use at exit: 0 bytes in 0 blocks
==20437==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==20437==
==20437== All heap blocks were freed -- no leaks are possible
==20437==
==20437== For counts of detected and suppressed errors, rerun with: -v
==20437== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Static Memory Errors - Options

- What options do you have?
- Valgrind's experimental SGCheck tool[4] sometimes helps (but not always)

```
valgrind --tool=exp-sgcheck ./my_program
```

- Recent versions of GCC and clang include instrumentation and checking (for more than just these errors)

```
gcc -fsanitize=bounds
gfortran -fcheck=bounds
```

- This can work quite nicely (requires a recent gcc)

```
$ cd /examples/7_static_error/c
$ make clean && make CFLAGS+=-fsanitize=bounds
gcc -fsanitize=bounds static_error.c -o static_error
$ ./static_error
static_error.c:8:4: runtime error: index 11 out of bounds for type 'int [10]'
static_error.c:10:8: runtime error: index 12 out of bounds for type 'int [10]'
b = 1473927512
static_error.c:13:8: runtime error: index 11 out of bounds for type 'int [10]'
c = 1473927512
a[7] >= 0
```

[4] http://valgrind.org/docs/manual/sg-manual.html

Best Practices

Valgrind Best Practices

- Use it often (it's easier than most diagnostic tools)
- Use `-O0 -g` for better diagnostic information
- Fix errors in the order that they occur (just like normal debugging)
- Don't ignore definite leaks
- Just like with warnings, keep your code as valgrind-clean as possible, so that the tool continues to be useful as you add new features
- It's no substitute for careful reasoning about your code, as you write it.

Other Best Practices

- Valgrind/Memcheck won't help with everything
- Use as many warning flags as you can
- Use modern instrumentation tools while debugging (easy with new versions of gcc/gfortran/clang!)
- Build and run code often (consider test-driven design)
- Use version control (such as git)
- Fortran: use Fortran 90, and don't use implicit interfaces (use modules)
- Read the documentation at your leisure. `http://valgrind.org/`

Thank you for your attention!