ECE297 Quick Start Guide
Valgrind

"Trust but verify."
– Ronald Reagan

1 Introduction

`valgrind`[1] is an open-source framework for debugging and profiling programs released under the GNU General Public Licence version 2. It was originally written by Julian Seward, who won a Google-O'Reilly Open Source Award for his work. A number of contributors have worked on it since.

This document explains the use of valgrind's `memcheck` tool for detecting memory errors.

1.1 Why use Valgrind?

A traditional debugger like gdb is a powerful tool to find problems in your program. You can set breakpoints, single-step, inspect variables, call functions, etc to watch the flow of your program. However, the debugger is not the ideal tool for all cases. When you have a segmentation fault, for instance, a debugger will show you the line of code where the fault occurred, the value of local variables, and the call stack that got you there. This still leaves you to try and deduce *why* the memory access is invalid. Sometimes, for instance dereferencing a null pointer, this is fairly trivial. Often it is not. Valgrind helps you by telling you why the access is invalid: are you trying to read beyond the end of an array? Are still using data in your program that you already `deleted`? Are you accessing memory through a pointer that was never initialized? These kinds of memory problems are some of the most difficult issues to debug in a C++ program, particularly if your program does not seg fault right away, but instead *corrupts* memory by changing memory locations that you never intended to change.

1.2 How it works (briefly)

When debugging, you typically instruct the compiler to output some extra information called a symbol table (using the `-g` option in g++). The symbol table gives the location of all the functions and variables inside your program. That way, the debugger (or other software) can find what the variable name is for a given address and what line of code a given instruction comes from. A traditional debugger like gdb loads your program and the symbol table, and then runs it as-is. The process is as fast or nearly so compared to running without the debugger, but what you are able to do is limited by the hardware debugging support in your processor.

Valgrind takes a different approach. It actually modifies your program, adding *instrumentation* (in Valgrind called a "tool") that gathers extra information when your program is run. In our case, the tool of interest is called `memcheck` which adds code to check the validity of *every memory access* done by the program. It also tracks uses of the dynamic memory management functions `malloc` and `free`. As you will see later, these C standard library calls are used by C++'s `new` and `delete`

[1] http://www.valgrind.org

operators. Clearly, this is a powerful tool when you have memory problems. The instrumentation added by valgrind does slow down your program, and it usually runs about 10x slower than usual.

1.3 Getting valgrind

The tool is already installed on the ECE machines. Since it is open-source, you may download it yourself at no cost from the website. In Linux, you can simply use your package management tool (`apt-get`, `yum`, `rpm`, or similar) to download and build automatically. For Mac OS, you can install from source or use the `fink` package management tool similarly to Linux. Unfortunately, the tool is not available for Windows and is not planned to be released due to the complexity of porting.

2 Memory errors

2.1 Leaks

If memory is allocated with `new` and then "forgotten about" or lost, it is called a memory leak. This is hazardous to the system because memory is a finite resource. If a program will run for a long time (eg. an operating system, web server, etc.) it will eventually crash for lack of memory. To avoid memory leaks, you must ensure that you always have a copy of the pointer returned when you create something using `new`. As soon as you are done with the memory, it should be freed using `delete`.

```
void foo () {
   int *p = new int;
} // No delete and pointer p is gone --> memory leaked!
```

Note that when a program ends, the operating system automatically reclaims all the program's memory. This means even leaked memory is reclaimed when a program ends, but you should still avoid memory leaks, as your program will be wasting memory for as long as it is running if it leaks memory.

2.2 Double free

The opposite of a memory leak is a "double free" or attempting to `delete` memory that has already been `delete`d. This usually causes your program to crash with an error message from the operating system. Each block that is allocated (with `new`) must be freed exactly once (with `delete`).

```
int *p = new int;
delete p;   // Good --> avoid memory leak.
delete p;   // Double delete!
```

2.3 Invalid read/write

Reading or writing memory outside of the region you have allocated may cause a segmentation fault; a segmentation fault occurs when the operating system detects that you are trying to access

memory not owned by your program. However, it is also possible that the invalid memory access occurs within the range of memory owned by your own program – in this case the memory access will not be caught by the operating system and will not trigger a segmentation fault. The invalid memory access will, however, lead to unpredictable results as accesses you think are occurring to one variable are actually going to some other variable in memory. In fact, writing one or two elements past the end of an array often does not cause a segfault but it will unintentionally overwrite other variables. Fortunately, Valgrind's `memcheck` tool can find these situations.

```
int *myArray = new int[5];  // Has entries from [0] to [4].
myArray[5] = 12;           // Out of bounds write. Wrote 12 to unknown spot in memory!
int badVal = myArray[6];  // Out of bounds read. Set badVal to an unknown value!
```

2.4 Uninitialized pointer

One of the dangerous things about pointers is that if they are not properly initialized they will point to a random location. This will often cause a segmentation fault if you try to dereference the pointer, but it can also lead to you reading or writing unknown data. Tracing the origin of the problem can be difficult if the pointer is passed around and copied through a variety of functions and classes. Valgrind can help you identify operations on uninitialized pointers and trace their origins.

```
int *p;   // Not initialized
*p = 20;  // Just wrote 20 to some random spot in memory!
```

3 Running Valgrind

Once Valgrind is installed and added to your `PATH` environment variable, running it is simple. You first compile your program with debug symbols enabled (`-g` option in g++) because Valgrind needs these symbols to tell you the line numbers and function names. Then, you can run valgrind from a command prompt, giving it its own options first followed by your program name and the program options:

```
> valgrind [valgrind-options] <program-name> [program-options]
```

For instance, if you wanted to run valgrind on the program of lab 3, which is called `rnet` and which does not take any command-line arguments you would run:

```
> valgrind ./rnet
```

When the current directory (`.`) is not in your path - as on the ECF machines - you need to specify `./rnet` as shown above, instead of just `rnet` so valgrind finds your program. Valgrind will then insert the desired error checking instrumentation into `rnet` (in this case the tool we want, `memcheck`, is the default), and run it. Your program will show its normal output. Valgrind will by default show its (error-checking) output on *standard error*. Table 1 lists some of valgrind's options.

`--log-file=`*fn*	Saves Valgrind output to file *fn*
`--leak-check=full`	Provide detailed information on memory leaks instead of just summary
`--dsymutil=yes`	Required for Mac OS X to provide correct debug info
`--read-var-info=yes`	Reads more-detailed variable information to provide more informative messages
`--show-reachable=yes`	Shows all "reachable" blocks. These are blocks of memory allocated with `new` that are not leaked (there are still pointers to them) but which were not deleted before your program exited.
`--track-origins=yes`	Tracks the origin of (line of code that created) uninitialized values at the cost of high memory usage and very slow runs

Table 1: Useful Valgrind arguments

4 Example program

4.1 Source code and output

This example program has code to create an array holding the first N values in two different integer sequences: the factorial sequence and the Fibonacci sequence. It contains a number of deliberately-inserted memory errors (invalid accesses, leaks, and double-frees) to illustrate the use of Valgrind to trace such errors. The program source is:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5
6  // Allocates an array of N ints and fills it with the Fibonacci sequence
7  // The sequence is defined by F[i] = F[i-1] + F[i-2] with F[0]=F[1]=1
8  int* makeFibonacciArray(int N)
9  {
10     int *fibArray = new int[N];
11     fibArray[0]=1;
12     fibArray[1]=1;
13     for(int j=2;j<N;++j)
14         fibArray[j] = fibArray[j-1]+fibArray[j-2];
15     return fibArray;
16 }
17
18
19 // Allocates an array of N ints and fills it with the sequence N!
20 // The sequence is defined by F[i] = i*F[i-1] with F[0]=1
21 int* makeFactorialArray(int N)
22 {
23     int *facArray = new int[5];
24
25     facArray[0]=1;
26     for(int i=1; i< N; ++i)
27         facArray [i] = i*facArray[i-1];
28     return facArray;
29 }
```

```
30
31
32  // Prints N elements of a sequence, giving a title line with the sequence name
33  // and N. Then prints one element per line, with an indent of 2 spaces.
34  void printSequence(string name,int* sequence,int N)
35  {
36      cout << "First " << N << " elements of sequence \"" << name << "\" are:\n";
37      for(int i=0;i<N;++i)
38          cout << "  " << sequence[i] << endl;
39  }
40
41
42  int main() {
43
44      int *seq = makeFibonacciArray(10);
45
46      printSequence("fibonacci",seq,10);
47
48      seq = makeFactorialArray(6);
49
50      printSequence("factorial",seq,6);
51
52      int *fac = seq;
53
54      int *yet_another = makeFactorialArray(5);
55
56      delete[] fac;
57      delete[] seq;
58  }
```

See how many errors (including leaks) you can spot just by inspecting the code. Now compile the program (with the -g option to add debugging information) and run it:

```
> g++ -g vgexample.cpp -o vgexample
> ./vgexample
```

The program will print out two sequences, and then crash with a memory error:

```
First 10 elements of sequence "fibonacci" are:
  1
  1
  2
  3
  5
  8
  13
  21
  34
  55
First 6 elements of sequence "factorial" are:
  1
  1
  2
```

```
  6
  24
  120
*** glibc detected *** ./vgexample: double free or corruption (fasttop): 0x09fcf058 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(+0x6e341)[0x17e341]
/lib/tls/i686/cmov/libc.so.6(+0x6fb98)[0x17fb98]
/lib/tls/i686/cmov/libc.so.6(cfree+0x6d)[0x182c7d]
/usr/lib/libstdc++.so.6(_ZdlPv+0x21)[0xe8b741]
/usr/lib/libstdc++.so.6(_ZdaPv+0x1d)[0xe8b79d]
./vgexample[0x8048cb4]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0x126bd6]
./vgexample[0x8048911]
```

The line `*** glibc detected *** ./vgexample: double free or corruption` indicates that you have a memory fault. Looking at the stack trace below it, you can see that the error occurs deep in the C++ libraries, which are called from your `main` routine. You may be able to see the problem if you carefully inspect the code, but if not, move on to the next section and try the debugger.

4.2 Use the debugger

Now try using the debugger. You may do this either through NetBeans, or by running

```
> gdb vgexample
```

Having done that, you should see it fail with a message similar to the following (exact details may vary by library implementation):

```
vgexample(26327) malloc: *** error for object 0x1001000e0: pointer being freed was
    not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap
```

Now view the Call Stack pane in NetBeans (or run a backtrace, `bt`, if using command-line GDB), or to find the error source. It should be on line 57. If you look up just a few lines, you should see that the pointers `fac` and `seq` are identical so this is a double-delete situation. Typically, it will not be nearly so easy to tell. The fix is simple: eliminate one of the `delete[]` calls. Now the program runs perfectly. Or does it? Let's see what Valgrind finds.

5 Running Valgrind on your program

First, build the program with debugging information included, and ensure the executable file is called `vgexample`

```
g++ -g vgexample.cpp -o vgexample
```

Next run valgrind on this program.

```
valgrind --leak-check=full ./vgexample
```

This will run Valgrind with an option to output extra information about memory leaks and all output going to the terminal. **For those using Mac OS: you will need to add the option `-dsymutil=yes` between `valgrind` and `vgexample`.**

6 Interpreting Valgrind output

Before looking at sample output, we should note that Valgrind output may vary subtly across platforms. The output below was produced on an ECF machine running Ubuntu; on other platforms, there may be some differences in appearance.

Now we examine the Valgrind output line by line to see what it can tell us. First, it starts with a banner. Notice that all lines are prefixed by `==#######==` which is the Process ID (PID). If we had multiple processes (programs) running at once, that would be helpful. We are debugging only one program, however, so this process ID information is not very useful to us except that it shows us the output lines generated by valgrind, and not by our program.

```
==3147== Memcheck, a memory error detector
==3147== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==3147== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==3147== Command: ./vgexample
==3147==
```

Next we see some of the regular output from vgexample. It looks like vgexample has loaded the array storing the Fibonacci series and printed it out with no memory issues.

```
First 10 elements of sequence "fibonacci" are:
  1
  1
  2
  3
  5
  8
  13
  21
  34
  55
```

Next, we see that there is an invalid write (writing to memory we don't own). Note that the debugger missed this because it didn't cause a segmentation fault. Sometimes when you happen to hit memory "owned" by your program an invalid read/write will not cause a fault. It will, however, lead to unpredictable behavior that is hard to trace.

```
==3147== Invalid write of size 4
==3147==    at 0x8048A5B: makeFactorialArray(int) (vgexample.cpp:27)
==3147==    by 0x8048BC1: main (vgexample.cpp:48)
==3147==  Address 0x42d30dc is 0 bytes after a block of size 20 alloc'd
```

```
==3147==    at 0x402532E: operator new[](unsigned int) (vg_replace_malloc.c:299)
==3147==    by 0x8048A2A: makeFactorialArray(int) (vgexample.cpp:23)
==3147==    by 0x8048BC1: main (vgexample.cpp:48)
==3147==
```

Not only does valgrind tell us where the invalid write happens, but it also gives the call stack that got us there (innermost first, just like in the debugger). The bad memory write is occuring on line 27 of vgexample.cpp, in the `makeFactorialArray` subroutine. This line is

```
facArray [i] = i*facArray[i-1];
```

Now we know that for some value of `i`, `facArray[i]` is not within the space we allocated to the array.

Valgrind also gives some extra information about this invalid write that may be helpful. It tells us that the address we were trying to write to is right after (0 bytes after) a block of 20 bytes that was allocated on line 23 of vgexample.cpp. That line is:

```
int *facArray = new int[5];
```

facArray is 5 integers in size, so it makes sense that it was allocated 20 bytes (5 * 4 bytes per int). Since we are trying to access the entry right after this array, it means we are trying to access facArray[6]. Sure enough, the main routine is calling `makeFacorialArray` with N=6

```
seq = makeFactorialArray(6);
```

but we only allocated 5 entries in the `new[]` call that allocated space for `facArray`. We have written to memory outside the array – some other variable has been unintentionally changed. To fix this problem we should allocate facArray to have `N` entries instead of 5 entries.

Next, vgexample outputs some more of its regular output, printing out most of the factorial sequence:

```
First 6 elements of sequence "factorial" are:
  1
  1
  2
  6
  24
```

There is then another memory fault, followed by the last line of the factorial sequence output ($120 = 5!$):

```
==3147== Invalid read of size 4
==3147==    at 0x8048ADE: printSequence(std::string, int*, int) (vgexample.cpp:38)
==3147==    by 0x8048C27: main (vgexample.cpp:50)
==3147==  Address 0x42d30dc is 0 bytes after a block of size 20 alloc'd
==3147==    at 0x402532E: operator new[](unsigned int) (vg_replace_malloc.c:299)
==3147==    by 0x8048A2A: makeFactorialArray(int) (vgexample.cpp:23)
==3147==    by 0x8048BC1: main (vgexample.cpp:48)
==3147==
  120
```

Reading the valgrind output carefully, you can see that this time we are reading from a memory location we did not allocate, in line 38 of the PrintSequence subroutine, which is:

```
cout << "  " << sequence[i] << endl;
```

The memory we are reading is right after (0 bytes after) a block of memory allocated on line 23 of the program, in makeFactorialArray. This array is once again `facArray` a pointer to which has been passed in as `sequence`. We allocated this array to have only 5 entries, meaning it has entries 0 to 4, but we are trying to output `sequence[5]`, an invalid read.

Next valgrind detects the double delete at the end of the program:

```
==3147== Invalid free() / delete / delete[]
==3147==    at 0x40244D3: operator delete[](void*) (vg_replace_malloc.c:409)
==3147==    by 0x8048CB3: main (vgexample.cpp:57)
==3147==  Address 0x42d30c8 is 0 bytes inside a block of size 20 free'd
==3147==    at 0x40244D3: operator delete[](void*) (vg_replace_malloc.c:409)
==3147==    by 0x8048CA0: main (vgexample.cpp:56)
==3147==
```

We were earlier able to diagnose this problem in the debugger, but notice that valgrind's output is more explicit and easier to understand. Valgrind tells us that we are trying to `delete[]` a memory block on line 57, but that block was already deleted on line 56. Those lines are:

```
delete[] fac;
delete[] seq;
```

Reading the code in `main` carefully, you can see that `fac` and `seq` point at the same allocated memory block, so we have deleted it twice. We should remove one of these two lines from the program to fix the bug.

Finally, the program finishes. Valgrind then checks for any memory that was allocated (with `new`) and never deleted, and prints a report on this memory "in use at exit". If a block of memory is both in use at exit and there is no pointer to it, we have a memory leak: memory that the program could not possibly delete.

```
==3147==
==3147== HEAP SUMMARY:
==3147==     in use at exit: 60 bytes in 2 blocks
==3147==   total heap usage: 5 allocs, 4 frees, 124 bytes allocated
==3147==
==3147== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==3147==    at 0x402532E: operator new[](unsigned int) (vg_replace_malloc.c:299)
==3147==    by 0x8048A2A: makeFactorialArray(int) (vgexample.cpp:23)
==3147==    by 0x8048C6B: main (vgexample.cpp:54)
==3147==
==3147== 40 bytes in 1 blocks are definitely lost in loss record 2 of 2
==3147==    at 0x402532E: operator new[](unsigned int) (vg_replace_malloc.c:299)
==3147==    by 0x80489B7: makeFibonacciArray(int) (vgexample.cpp:10)
==3147==    by 0x8048B3D: main (vgexample.cpp:44)
==3147==
==3147== LEAK SUMMARY:
```

```
==3147==    definitely lost: 60 bytes in 2 blocks
==3147==    indirectly lost: 0 bytes in 0 blocks
==3147==      possibly lost: 0 bytes in 0 blocks
==3147==    still reachable: 0 bytes in 0 blocks
==3147==         suppressed: 0 bytes in 0 blocks
```

Our program has leaked 2 blocks of memory totaling 60 bytes. The exact sequence of subroutines called to allocate each block is given to us to help us figure out where the memory was allocated with `new`. The first leaked block is 20 bytes in size, and was created by line 54 of main calling makeFactorialArray, where the memory was allocated on line 23 (facArray). The problem is now fairly obvious – we store a pointer to the allocated memory in `yetanother`, but never delete it. The other memory leak comes from line 44 in main – we allocate an array by calling makeFibonacciArray, but overwrite the pointer to it that we stored in variable `seq`. To fix these bugs, we need to add two additional `delete[]` calls at the appropriate points in main.

Note that if you compiled vgexample with the `-Wall` g++ option (this option is also set when you choose "More Warnings" in your NetBeans compiler options), the compiler itself gives a warning about the fact that we never use the (read) the variable `yetanother`. This is a good example of how turning on all compiler warnings can save you time by pointing out strange things in your code at compile time.

Your lab programs should not have any "definitely lost" blocks; you will lose marks if they do.

7 Other Valgrind Tools

The memcheck tool on which this tutorial focused is just part of the valgrind framework; other useful tools are listed in Table 2. The purpose of all some of these tools won't be clear to you at this stage of your programming career, but will be by the end of your undergraduate degree.

memcheck	Checks for errors in memory allocation and access
cachegrind	Profiles cache utilization to find the source of performance problems
helgrind	Used for debugging races in multi-threaded programs
DRD	Similar to helgrind, uses less memory
massif	Provides information on heap (new) memory allocation

Table 2: Selected Valgrind tools and their uses