Valgrind tutorial*

Lena Olson
Department of Computer Sciences, University of Wisconsin-Madison
http://pages.cs.wisc.edu/~lena/

Last updated: 16 March 2011

Although C is a very useful and powerful language, it can be hard to debug. A particular problem that you have probably encountered at some point is memory errors. We have already talked about gdb, which can be a helpful resource if your program consistently crashes or outputs a wrong result. However, sometimes you suspect that the problem you are having is due to a memory error, but it does not cause a segfault and you do not want to set a lot of breakpoints and step through in gdb. Another common problem you might encounter is a program with a memory leak: somewhere, you call `malloc` but never call `free`. Valgrind is a program that will help you fix both problems.

To invoke it on an executable called `a.out`, you simply run the following command (with any arguments your program might need).

```
% valgrind ./a.out
```

As when using gdb, you will want to make sure to compile your program with the flag `-g`, so that you can see line numbers in the output. You may also wish to debug with optimizations turned off (`-O0`), since if you have them on, line numbers may be inaccurate and you may occasionally encounter false alarms.

Example 1: reading/writing past the end of an array. One common mistake is accessing elements past the end of an array. Your C program might segfault, or it might continue running, producing a result which is correct or incorrect – sometimes with results varying between executions. This makes it hard to locate this kind of problem. Here is how you would use valgrind to find the bug:

```
#include <stdlib.h>

int main(void)
{
  int i, n = 10;
  int *a = malloc ((size_t)n * sizeof(int));
```

*LaTeX conversion of the web tutorial at http://pages.cs.wisc.edu/~lena/valgrind.php.

```
  if (!a)
  {
    /* malloc failed */
    return 1;
  }

  for (i = 0; i <= n; i++)
  {
    a[i] = i;
  }
  free(a);

  return 0;
}
```

Compile the program and run valgrind as following:

```
% clang -Weverything -Wextra -pedantic -g -O0 example1.c -o example1
% valgrind ./example1
```

Output:

```
==16370== Memcheck, a memory error detector
==16370== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16370== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==16370== Command: ./example1
==16370==
==16370== Invalid write of size 4
==16370==    at 0x4005F2: main (example1.c:15)
==16370==  Address 0x51f5068 is 0 bytes after a block of size 40 alloc'd
==16370==    at 0x4C28C50: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-
==16370==    by 0x4005B5: main (example1.c:6)
==16370==
==16370==
==16370== HEAP SUMMARY:
==16370==     in use at exit: 0 bytes in 0 blocks
==16370==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==16370==
==16370== All heap blocks were freed -- no leaks are possible
==16370==
==16370== For counts of detected and suppressed errors, rerun with: -v
==16370== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Your output might be slightly different, depending on the machine and version of valgrind and libraries installed, but should include the same types of errors. If you examine the output, you will see that there is 1 error listed (you do not need to worry about suppressed errors). Valgrind prints what the error was (**`Invalid write of size 4`**) as well as the stack. It also lists the file, function and line where this array was malloc'd.

Example 2: reading uninitialized memory. Another common problem is forgetting to initialize a variable or array before using it.

```
#include <stdio.h>

int main(void)
{
  int i, n = 9;
  int a[10];
  for (i = 0; i < n; i++)
  {
    a[i] = i;
  }

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

  return 0;
}
```

Compile the program and run valgrind as following:

```
% clang -Weverything -Wextra -pedantic -g -O0 example2.c -o example2
% valgrind ./example2
```

Output:

```
==16376== Memcheck, a memory error detector
==16376== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16376== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==16376== Command: ./example2
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E81FA9: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Use of uninitialised value of size 8
==16376==    at 0x4E7E13B: _itoa_word (_itoa.c:179)
==16376==    by 0x4E82249: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E7E145: _itoa_word (_itoa.c:179)
```

```
==16376==    by 0x4E82249: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E822BC: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E8206E: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E8268A: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E820BE: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
==16376== Conditional jump or move depends on uninitialised value(s)
==16376==    at 0x4E820FE: vfprintf (vfprintf.c:1641)
==16376==    by 0x4E88E78: printf (printf.c:33)
==16376==    by 0x4005AF: main (example2.c:14)
==16376==
0 1 2 3 4 5 6 7 8 0
==16376==
==16376== HEAP SUMMARY:
==16376==     in use at exit: 0 bytes in 0 blocks
==16376==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==16376==
==16376== All heap blocks were freed -- no leaks are possible
==16376==
==16376== For counts of detected and suppressed errors, rerun with: -v
==16376== Use --track-origins=yes to see where uninitialised values come from
==16376== ERROR SUMMARY: 8 errors from 8 contexts (suppressed: 0 from 0)
```

Observe that the output of the program and the output of valgrind are interleaved; to get around that, it is handy to redirect the output to a separate file. (Use the option `--log-file=thefile` if you want this.) This time the errors reported are for uninitialized values, and valgrind indicates where the access takes place (line 14 of example2.c). If you run with the option `--track-origins=yes`, valgrind will give additional information about where the uninitialized values came from.

Example 3: memory leaks. Valgrind includes an option to check for memory leaks. With no option given, it will list a heap summary where it will say if there is any memory that has been allocated but not freed. If you use the option `--leak-check=full` it will give more information.

```
#include <stdlib.h>

int main (void)
{
  int i;
  int *a = NULL;

  for (i = 0; i < 10; i++)
  {
    a = malloc (sizeof(int) * 100);
  }
  free (a);

  return 0;
}
```

Compile the program and run valgrind as following:

```
% clang -Weverything -Wextra -pedantic -g -O0 example3.c -o example3
% valgrind --leak-check=full ./example3
```

Output:

```
==16382== Memcheck, a memory error detector
==16382== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16382== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==16382== Command: ./example3
==16382==
==16382==
==16382== HEAP SUMMARY:
==16382==     in use at exit: 3,600 bytes in 9 blocks
==16382==   total heap usage: 10 allocs, 1 frees, 4,000 bytes allocated
==16382==
==16382== 3,600 bytes in 9 blocks are definitely lost in loss record 1 of 1
==16382==    at 0x4C28C50: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-
==16382==    by 0x4005C9: main (example3.c:10)
==16382==
==16382== LEAK SUMMARY:
==16382==    definitely lost: 3,600 bytes in 9 blocks
==16382==    indirectly lost: 0 bytes in 0 blocks
==16382==      possibly lost: 0 bytes in 0 blocks
==16382==    still reachable: 0 bytes in 0 blocks
==16382==         suppressed: 0 bytes in 0 blocks
```

```
==16382==
==16382== For counts of detected and suppressed errors, rerun with: -v
==16382== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

If you see leaks indicated as still reachable, this generally does not indicate a serious problem since the memory was probably still in use at the end of the program. However, any leaks listed as "definitely lost" should be fixed (as should ones listed "indirectly lost" or "possibly lost" – "indirectly lost" will happen if you do something like free the root node of a tree but not the rest of it, and "possibly lost" generally indicates the memory is actually lost). An example of where you might run into an example like the one above is if you have a function that allocates a buffer (perhaps to store a string) and returns it, but the caller never frees the memory after it is finished. If a program like that runs for a long time, it will allocate a lot of memory that it does not need.

What valgrind is NOT

Although valgrind is an extremely useful program, it will not miraculously tell you about every memory bug in your program. There are several limitations that you should keep in mind. It does not do bounds checking on stack/static arrays (those not allocated with malloc), so it is possible to have a program that behaves badly while not generating any valgrind errors. For example:

```
#include <stdio.h>

int main (void)
{
  int i;
  int x = 0;
  int a[10];
  for (i = 0; i < 11; i++)
  {
    a[i] = i;
  }

  printf ("a[1] is %d.  ", a[1]);
  printf ("x is %d\n", x);

  return 0;
}
```

This program has a memory error, resulting in the value of x being 10 at the end rather than 0. However, valgrind will not report any errors.

Compile[1] the program and run valgrind as following:

```
% gcc -Wall -Wextra -pedantic -g -O0 example3.c -o example4
```

[1] In this example we use a different C compiler – gcc.

```
% valgrind ./example4
```

Output:

```
==16390== Memcheck, a memory error detector
==16390== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16390== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==16390== Command: ./example4
==16390==
a[1] is 1.  x is 10
==16390==
==16390== HEAP SUMMARY:
==16390==     in use at exit: 0 bytes in 0 blocks
==16390==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==16390==
==16390== All heap blocks were freed -- no leaks are possible
==16390==
==16390== For counts of detected and suppressed errors, rerun with: -v
==16390== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

A more serious limitation that you will encounter is that valgrind checks programs *dynamically* – that is, it checks during actual program execution whether any leaks actually occurred for that execution. This means that if you run valgrind on a particular set of inputs that does not cause any bad memory accesses or memory to be leaked, valgrind will not report any errors, even though your program does contain bugs. As an example:

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

int main(void)
{
  char *str = malloc((size_t) 10);

  gets(str);
  printf("%s\n",str);

  return 0;
}
```

This program has a bug: if the user inputs a long string, the buffer `str` will overflow. Please note that you should never, ever use `gets`, for exactly this reason. If you run this program through valgrind, you will get a memory error if you type in a string longer than 10 characters. However, if you type in a shorter string, valgrind will report no errors, even though the program is buggy! If you want to be reasonably sure that you are catching all memory bugs, you should run valgrind on a variety of inputs, especially corner cases, as those are where you are most likely to have made a mistake like accessing past the bounds of an array.

When fixing errors, it is a good idea to start at the top; fixing an error that occurs earlier is likely to eliminate a lot of later errors as well.

Once in a great while you may encounter a false positive – an error even though there is nothing wrong with your program. However, in the vast majority of cases, any error reported is real and you should fix it. Be very wary about dismissing an error as a ”false positive,” since it is much more likely that you have made a mistake.

One final thing to note about valgrind is that your programs will take longer to execute (like 20 to 30 times as long), and will also use more memory.

More information

If you are curious about valgrind, you can check the valgrind website, especially the FAQ.