

The Valgrind Quick Start Guide

Release 3.2.0 7 June 2006

Copyright © 2000-2006 Valgrind Developers

Email: valgrind@valgrind.org

Table of Contents

The Valgrind Quick Start Guide	1
1. Introduction	1
2. Preparing your program	1
3. Running your program under Memcheck	1
4. Interpreting Memcheck's output	1
5. Caveats	3
6. More information	3

The Valgrind Quick Start Guide

1. Introduction

The Valgrind distribution has multiple tools. The most popular is the memory checking tool (called Memcheck) which can detect many common memory errors such as:

- touching memory you shouldn't (eg. overrunning heap block boundaries);
- using values before they have been initialized;
- incorrect freeing of memory, such as double-freeing heap blocks;
- memory leaks.

What follows is the minimum information you need to start detecting memory errors in your program with Memcheck. Note that this guide applies to Valgrind version 2.4.0 and later; some of the information is not quite right for earlier versions.

2. Preparing your program

Compile your program with `-g` to include debugging information so that Memcheck's error messages include exact line numbers. Using `-O0` is also a good idea, if you can tolerate the slowdown. With `-O1` line numbers in error messages can be inaccurate, although generally speaking Memchecking code compiled at `-O1` works fairly well. Use of `-O2` and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist.

3. Running your program under Memcheck

If you normally run your program like this:

```
myprog arg1 arg2
```

Use this command line:

```
valgrind --leak-check=yes myprog arg1 arg2
```

Memcheck is the default tool. The `--leak-check` option turns on the detailed memory leak detector.

Your program will run much slower (eg. 20 to 30 times) than normal, and use a lot more memory. Memcheck will issue messages about memory errors and leaks that it detects.

4. Interpreting Memcheck's output

Here's an example C program with a memory error and a memory leak.

```
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;          // problem 1: heap block overrun
}                      // problem 2: memory leak -- x not freed

int main(void)
{
    f();
    return 0;
}
```

Most error messages look like the following, which describes problem 1, the heap block overrun:

```
==19182== Invalid write of size 4
==19182==   at 0x804838F: f (example.c:6)
==19182==   by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==   at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==   by 0x8048385: f (example.c:5)
==19182==   by 0x80483AB: main (example.c:11)
```

Things to notice:

- There is a lot of information in each error message; read it carefully.
- The 19182 is the process ID; it's usually unimportant.
- The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling you where the problem occurred. Stack traces can get quite large, and be confusing, especially if you are using the C++ STL. Reading them from the bottom up can help. If the stack trace is not big enough, use the `--num-callers` option to make it bigger.
- The code addresses (eg. 0x804838F) are usually unimportant, but occasionally crucial for tracking down weirder bugs.
- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with `malloc()` on line 5 of `example.c`.

It's worth fixing errors in the order they are reported, as later errors can be caused by earlier errors.

Memory leak messages look like this:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==   at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==   by 0x8048385: f (a.c:5)
==19182==   by 0x80483AB: main (a.c:11)
```

The stack trace tells you where the leaked memory was allocated. Memcheck cannot tell you why the memory leaked, unfortunately. (Ignore the "vg_replace_malloc.c", that's an implementation detail.)

There are several kinds of leaks; the two most important categories are:

- "definitely lost": your program is leaking memory -- fix it!
- "probably lost": your program is leaking memory, unless you're doing funny things with pointers (such as moving them to point to the middle of a heap block).

If you don't understand an error message, please consult [Explanation of error messages from Memcheck](#) in the [Valgrind User Manual](#) which has examples of all the error messages Memcheck produces.

5. Caveats

Memcheck is not perfect; it occasionally produces false positives, and there are mechanisms for suppressing these (see [Suppressing errors](#) in the [Valgrind User Manual](#)). However, it is typically right 99% of the time, so you should be wary of ignoring its error messages. After all, you wouldn't ignore warning messages produced by a compiler, right? The suppression mechanism is also useful if Memcheck is reporting errors in library code that you cannot change; the default suppression set hides a lot of these, but you may come across more.

Memcheck also cannot detect every memory error your program has. For example, it can't detect if you overrun the bounds of an array that is allocated statically or on the stack. But it should detect every error that could crash your program (eg. cause a segmentation fault).

6. More information

Please consult the [Valgrind FAQ](#) and the [Valgrind User Manual](#), which have much more information. Note that the other tools in the Valgrind distribution can be invoked with the `--tool` option.