# C Basics And Concepts
# Memory Leaks and Debugging with Valgrind

**— Report —**

Working group scientific computing
Department of informatics
Faculty of mathematics, informatics and natural sciences
University of Hamburg

| | |
|---|---|
| Written by: | Jula Menck |
| E-Mail: | 3menck@informatik.uni-hamburg.de |
| Matriculation number: | 5834957 |
| Bachelor degree course: | Software-System-Entwicklung |
| | |
| Advisor: | Konstantinos Chasapis |

Hamburg, 2014-06-19

# Contents

# 1 Introduction

This report is to present on memory leaks as well as show how to use Valgrind and its tool Memcheck to debug these. Firstly, it will cover what memory leaks are, how they are caused and will also elaborate on some other troublemakers concerning memory, which can also be debugged with Memcheck. Then it will introduce Valgrind and its memory error detecting tool Memcheck. Here Memcheck will be explained in further detail on how to use it and some examples will be given.

# 2 Memory Leaks

Memory leaks are mismanaged memory allocations. They are caused by heap areas that can no longer be freed, due to a lost pointer and are something every programmer using C has to be careful about. These leaks occur because C doesn't clean up after itself, unlike Java or C# with its inbuilt garbage collector. Memory leaks are hard to find because a program might work just fine for a while and then crash without apparent reason or simply slow down below acceptable levels. Sometimes this might be misconstrued as a hardware problem.

## 2.1 Troublemakers

Some memory troubles can be caused by using a pointer that has been allocated to a certain amount of bytes, wherein uninitialized data is contained or, worse yet, garbage data. Using this pointer can lead to unwanted or unexpectable issues with the code. An example gives the following code:

Listing 2.1: Code Example I

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

int main(void)
{
char *p;

char c = *p;

printf("\n␣[%c]\n",c);

return 0;
}
```

Another problem is overwriting memory. In this case the programmer chose to write more into the pointer than that pointer can possibly accept. This leads to the program writing the leftover data into other parts of the heap. If another pointer is pointing to that now (partly) rewritten heap space your program can yet again not work as intended.

The same problem can occur by overreading a pointer. In this case you read more bytes than were allocated to the pointer, thus also leading to unwanted issues. Again a code example can be seen below:

Listing 2.2: Code Example II

```
#include <stdlib.h>

int main()
{
char *ptr = (char *)malloc(10);
char name[20] ;
memcpy ( name,ptr,20);
}
```

## 2.2 Causes

There are several causes for memory leaks. Here are just a few. For one, one can lose a pointer through reassignment. Reassigning a pointer to point to a different heap area might make the previous heap area this pointer was targeting an orphan. This means it can no longer be accessed, because the only pointer was reassigned. A diagram is shown below to visualize a reassigning of the pointer "memoryArea" to a new address called "newArea", thus orphaning the old address of "memoryArea".

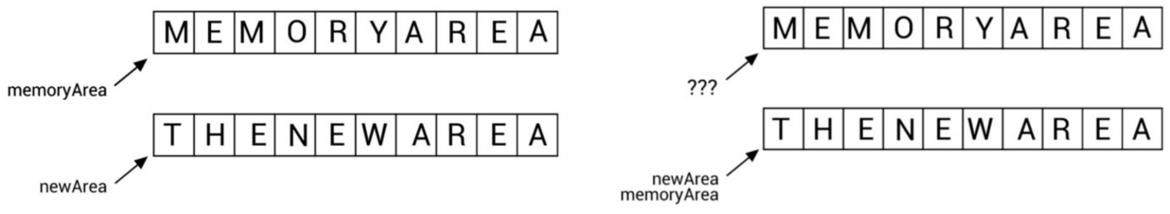

Figure 2.1: Reassignment of a Pointer

Another cause is freeing a parent block of a pointer. In a pointer structure you might have a child pointer. If one then chooses to free the parent pointer first, this child pointer will no longer be accessible, as it was within the parent structure. The code example below show a possible problem, wherein the parent pointer was freed without freeing the children pointers inside the array. The diagram is just to show a visualization of the array.

Listing 2.3: Code Example III

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    int z=3;
    int s=z+1;
    int **matrix ;
    int i,k;
```

```
        matrix=(int**)malloc(z*sizeof(void*));
        for (i=0;i<z;i++)
        {
         matrix[i]=(int*)malloc(s*sizeof(int));
     }
     free(matrix);
     return 0;
    }
```

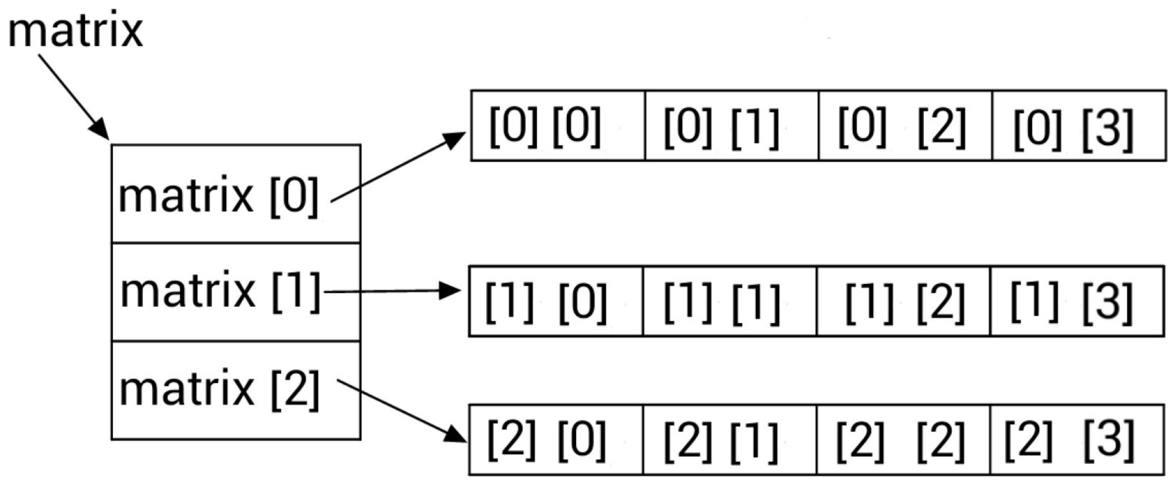

Figure 2.2: Parent and Child Pointers within an Array

Yet another cause is improper handling of return values. If a function returns a reference to a dynamically allocated memory it is the job of the calling function to keep track of the memory location. If it fails to do so you lose the address.

## 2.3 Tips

Some tips on avoiding memory related errors and leaks are as follows: One should always remember to use free() after malloc()It helps if one writes these two commands right after one another and then start programming what is to come in between, so as not to forget to write free() later on. Another thing that might be helpful is to not work with the original pointer, but instead use a copy of a pointer. This allows you to not lose the address of the pointer, even through reassignment, as one is not using the original. Also one should be careful about orphaning memory locations when (re)assigning pointer. like taking care to free a structured pointer from child to parent and not the other way around. One should also take care not to access null pointers and to handle returned references properly and not lose them.

# 3 Valgrind

## 3.1 Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. It is an open source software licensed under the GNU General Public License. It can be run on Linux, Mac or Android architecture, but will not run under Windows. It can be used for any program language, but some tools are predestined for certain program languages all the same. It is build in a modular architecture and includes six production-quality tools. These include among others a memory error detector - this being Memcheck - a cache and branch-prediction profiler, a heap profiler and several others. There are also three more experimental tools. Valgrind is a tool that will simulate every single instruction of your program including libraries and suppressions and is done on a synthetic core given by Valgrind, which is the reason that a program needs to be started with Valgrind attached to be able to debug it.

## 3.2 Memcheck

Memcheck is the error detection tool of Valgrind. It is also its standard tool. It is especially made for languages such as C, C++ and Fortran. While using it your program will run about 10-50 times slower. It will report errors it detects before executing the code. This allows the program to report on the error before possibly causing a crash with the buggy code. It also allows for suppressing errors, which can be useful as Memcheck sometimes sees errors in the system library codes. As one might not want to see these error messages you can have a suppression message written, which can then be copied into a suppression file for Memcheck.

## 3.3 Checks done by Memcheck

The Memcheck tool checks for several things. For one it tracks addressability per byte and initialization per bit, enabling it to detect the use of single uninitialized bits. It also keeps track of heap blocks allocated with malloc(), thus can detect false or missing frees. Furthermore it checks all reads and writes of memory in case of them overlapping. Lastly it performs definedness checks, allowing it to detect undefined value errors with bit-precision. This is done via shadow bits. Every single bit of data is shadows with these shadow bits, which ascertains whether the bit in question has a properly defined value at any given point in time.

## 3.4 Detections possible by Memcheck

Following are a variety of things Memcheck can detect with the checks it does. For one it can detect if memory is accessed which shouldn't be accessed. These are for example overrunning or underrunning heap blocks and accessing memory after it has been freed. It can also detect whether undefined values are being used. Furthermore it will let one know whether heap memory was incorrectly freed e.g. double-freeing heap blocks or different amounts of malloc vs. free. Also it is able to detect if there is an overlap of src and dst pointers in memcpy and related functions. Then of course it can also detect memory leaks in general.

## 3.5 Commands

The tools of Valgrind can be invoked via the following command: valgrind [valgrind-options] your-program [your-program-options]. As Memcheck is the default tool you can omit the "–tool=name" option for the valgrind-options. There is one thing one has to be careful about. This is the use of compiler optimization flags. Using "-o2" or "-o1" can sometimes report (missing) uninitialized value errors where there are none. Not using optimization might not be an option as it would be too slow. Instead using "-o" is a possible compromise while using Memcheck. Other Valgrind tools are unaffected by optimization flags. The following table is a summary of the commands explained above.

| **commands** | **description** |
|---|---|
| -g | Helps Valgrind match function to code, this has to be done on compiling |
| -v | Adds more detail to error message description (another -v adds yet more detail) |
| –num-callers | Makes stack traces longer |
| –track-origins=yes | See the origins of uninitialized values |
| –gen-suppressions=yes | Writes a suppression for each error message which you can copy into a suppression file |
| –read-var-info | gives more detailed description of any illegal addresses (may run slower) |

Table 3.1: Commands

## 3.6 Options

There are several important options one should know when using Memcheck. Using "-g" for example helps Valgrind match function to code, whereas it might otherwise not be able to give detailed enough info as to where the error is taking place. This option has to be done upon compiling. The "-v" command makes Memcheck more verbose;

it will give a more detailed error message description. Adding another "-v" will add yet more details. Using "-num-callers" will make the stack traces longer. The addition of "-track-origins=yes" will make one able to see the origins of uninitialized values. "-gen-suppressions=yes" will write a suppression for each error message which can then be copied into a suppression file. Lastly, using "-read-var-info" gives a more detailed description of any illegal addresses, but will make the program run even slower.

## 3.7 Commentary

The error messages are written into a commentary. This commentary is send to a specific location. The options of these locations are either the default, which is the stderr, or one can specify a file with "-log-file='filename'". Another option is to send it to a network socket via "-log-socket='IP-Address'". This will not work with a Host Name. On the end of that IP-Address one can listen in with the Valgrind-Listener.

## 3.8 Error Messages

As already mentioned above, Valgrind reports errors before the associated operation happens. The error messages it gives you include among others: "Invalid read/write of 'size number'". This message appears when an illegal read or write error occurs. Memcheck will give this message when it thinks that it shouldn't be accessed at that point. It will also give this message, if that block might have been freed already and inform you if the access might be off by one bit of the heap block. Another possible error message is: "Conditional jump or move depends on uninitialized value(s)". This appears if the program uses a value which hasn't been initialized or is undefined. Memcheck keeps track if you copy an undefined value, but will only comment on it, if it causes issues.

## 3.9 Leak Error Messages

The leak error messages are summarized in a leak summary with number of bytes and blocks. The possible messages here are: "Definitely lost". This tells one that the program is leaking memory, which needs to be fixed. "Indirectly lost" on the other hand informs one, that the program is leaking in a pointer-based structure (e.g. there is a parent node which is "definitely lost", making all the children "indirectly lost"). In this case fixing the "definitely lost" will most likely fix the "indirectly lost". There is also "possibly lost". This informs one, that the program is leaking memory, unless something inventive was done with pointers, like causing them to point to the middle of allocated blocks. Use of "–show-possibly-lost=no"will make Memcheck not inform you about those. "Still reachable" means that the program didn't free some memory it could have, but this being probably alright. This is quite common and often reasonable unless a lot of data is still in use. Use "-show-reachable=yes" if you want to see these errors. Lastly there is the message "suppressed". This tells one, that a leak error has been suppressed with a suppression file. Some suppressions are already written in the default suppression file and can be ignored.

# 4 Summary

Overall memory leaks occur by forgetting to free() space or by losing the pointer to a memory area before freeing that particular area. Leaks occur through a variety of reasons. Valgrind is an open source tool framework which allows one to debug a program. Memcheck is the standard tool of Valgrind and will help with detecting memory errors and leaks.

# Bibliography

[01] Valgrind manual. http://valgrind.org/docs/manual/manual.html.

[02] Pointers and memory leaks in c. http://www.ibm.com/developerworks/aix/library/au-toughgame/.

[03] Simple rules to avoid memory leaks in c. http://mousomer.wordpress.com/2010/11/03/simple-rules-to-avoid-memory-leaks-in-c/.

[04] How to detect memory leaks using memcheck tool for c or c++. http://www.thegeekstuff.com/2011/11/valgrind-memcheck/.

[05] Using valgrind to find memory leaks and invalid memory use. http://www.cprogramming.com/debugging/valgrind.html.

[06] Speicherverwaltung und fortgeschrittene pointer-themen. http://www.fh-kl.de/~guenter.biehl/lehrgebiete/c/c08.html.

[SJ05] Nethercote Nicholas Seward Julian. *Using Valgrind to detect undefined value errors with bit-precision.* Proceedings of the USENIX05 Annual Technical Conference., Anaheim, California, United States of America, 2005.

# List of Figures

# List of Tables

# List of Listings