Complete Valgrind Guide: Memory Debugging Tool

Introduction to Valgrind

What is Valgrind?

Valgrind is **not** a "value grinder" - it's named after the gate to Valhalla (Hall of the Slain) in Norse mythology. It is a powerful memory debugging tool capable of detecting many common memory-related errors and problems.[1]

Primary Uses:[2]

- Memory leak detection
- Invalid memory access detection
- Memory profiling
- In this guide, we focus on the **memcheck** feature

Basic Usage

Running Valgrind

Basic Syntax:[3]

```
valgrind executable [command line options]
```

Example:[4]

```
valgrind ./a.out 2022 -name "Sad Tijihba"
```

Understanding Valgrind Output

Sample Output (Clean Program):[5]

```
==4825== Memcheck, a memory error detector
==4825== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4825== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4825== Command: ./a.out
==4825==
[Your Program's Input/Output Here]
==4825==
==4825== HEAP SUMMARY:
==4825==     in use at exit: 0 bytes in 0 blocks
==4825==   total heap usage: 23 allocs, 23 frees, 1,376 bytes allocated
==4825==
```

[1] valgrind.pdf
[2] valgrind.pdf
[3] valgrind.pdf
[4] valgrind.pdf
[5] valgrind.pdf

```
==4825== All heap blocks were freed -- no leaks are possible
==4825==
==4825== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Key Points:

- The number `==4825==` is the process ID[6]
- `23 allocs, 23 frees` means all allocated memory was freed (perfect!)
- `0 errors` indicates no memory problems detected

Common Command-Line Options

1. Leak Check Options

`--leak-check=full`

Shows detailed information about memory leaks:[7][8]

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

`--show-leak-kinds=all`

Displays all types of memory leaks:[9]

```
valgrind --leak-check=full --show-leak-kinds=all ./myprogram
```

2. Tracking Uninitialized Values

`--track-origins=yes`

Tracks where uninitialized values come from:[10][11][12]

```
valgrind --track-origins=yes ./myprogram
```

Example Output:

```
==30384== Conditional jump or move depends on uninitialised value(s)
==30384==    at 0x400580: main (foo.c:10)
==30384==  Uninitialised value was created by a heap allocation
==30384==    at 0x4C2C66F: malloc (vg_replace_malloc.c:270)
==30384==    by 0x400551: func1 (foo.c:4)
```

This helps you find exactly where the uninitialized value was created.

[6] valgrind.pdf
[7] https://valgrind.org/docs/manual/quick-start.html
[8] https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks
[9] https://plus.tuni.fi/graderT/static/compcs300-compcs300-october-2024/lectures/trees/valgrind/tools.html
[10] https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks
[11] https://stackoverflow.com/questions/40810319/valgrind-warning-unknown-option-track-origins-yes
[12] https://gist.github.com/gaul/5306774

3. Output Control

`--log-file=filename`

Saves Valgrind output to a file:[13][14]

```
valgrind --log-file=output.txt ./myprogram
```

Using special patterns:

```
valgrind --log-file="valgrind-%p.log" ./myprogram
```

Where `%p` is replaced by the process ID.

`-q` or `--quiet`

Suppresses informational messages, shows only errors:[15]

```
valgrind -q --leak-check=full ./myprogram
```

4. Error Handling

`--error-exitcode=number`

Returns specified exit code when errors are found:[16]

```
valgrind --error-exitcode=99 ./myprogram
```

Useful for automated testing and CI/CD pipelines.

`--num-callers=number`

Controls stack trace depth (default: 12):[17]

```
valgrind --num-callers=20 ./myprogram
```

Memory Leak Categories

Valgrind classifies unfreed memory into four categories:[18][19][20]

1. Definitely Lost

Memory that is no longer accessible - **true memory leaks**[21]

[13] https://stackoverflow.com/questions/8355979/how-to-redirect-valgrinds-output-to-a-file
[14] https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks
[15] https://bytes.usc.edu/cs104/wiki/valgrind/
[16] https://stackoverflow.com/questions/76698927/why-is-valgrind-ignoring-my-error-exitcode-option
[17] https://stackoverflow.com/questions/11242795/how-to-get-the-full-call-stack-from-valgrind
[18] https://developers.redhat.com/blog/2021/04/23/valgrind-memcheck-different-ways-to-lose-your-memory
[19] http://web.stanford.edu/class/archive/cs/cs107/cs107.1262/resources/valgrind
[20] valgrind.pdf
[21] valgrind.pdf

2. Indirectly Lost

Memory accessible only through pointers in “definitely lost” blocks[22]

3. Possibly Lost

Memory accessible only via interior pointers (not pointing to the start)[23]

4. Still Reachable

Memory that wasn’t freed but is still accessible at program exit - not errors, but cleanup opportunities[24][25]

Practical Examples from PDF

Example 1: Linked List - Delete Without Freeing (Memory Leak)

Problematic Code:[26]

```
node *ldel(node *L, int x) {
    node *p;
    p = L;
    while (p->next) {
        if (p->next->data == x) {
            p->next = p->next->next;  // Memory leak! Node not freed
            return L;
        }
        if (p->next->data > x) break;
        p = p->next;
    }
    return L;
}
```

Valgrind Output:[27]

```
==9837== HEAP SUMMARY:
==9837==     in use at exit: 144 bytes in 9 blocks
==9837==   total heap usage: 10 allocs, 1 frees, 1,168 bytes allocated
==9837==
==9837== LEAK SUMMARY:
==9837==   definitely lost: 48 bytes in 3 blocks
==9837==   indirectly lost: 32 bytes in 2 blocks
```

[22] valgrind.pdf
[23] valgrind.pdf
[24] https://stackoverflow.com/questions/67040349/valgrind-gives-error-memory-still-reachable
[25] valgrind.pdf
[26] valgrind.pdf
[27] valgrind.pdf

```
==9837==      possibly lost: 0 bytes in 0 blocks
==9837==    still reachable: 64 bytes in 4 blocks
```

Analysis: The program allocated 10 blocks but freed only 1, resulting in definitely lost and indirectly lost memory.

Example 2: Linked List - Delete With Proper Freeing

Corrected Code:[28]

```
node *ldel(node *L, int x) {
    node *p, *q;
    p = L;
    while (p->next) {
        if (p->next->data == x) {
            q = p->next;
            p->next = q->next;
            free(q);  // Properly freed!
            return L;
        }
        if (p->next->data > x) break;
        p = p->next;
    }
    return L;
}
```

Valgrind Output:[29]

```
==10012== HEAP SUMMARY:
==10012==     in use at exit: 64 bytes in 4 blocks
==10012==   total heap usage: 10 allocs, 6 frees, 1,168 bytes allocated
==10012==
==10012== LEAK SUMMARY:
==10012==   definitely lost: 0 bytes in 0 blocks
==10012==   indirectly lost: 0 bytes in 0 blocks
==10012==     possibly lost: 0 bytes in 0 blocks
==10012==    still reachable: 64 bytes in 4 blocks
```

Analysis: No definitely/indirectly lost memory! The remaining blocks are “still reachable” (the list itself).

---

[28] valgrind.pdf
[29] valgrind.pdf

Example 3: Freeing All Nodes - Perfect Cleanup

Complete Cleanup Code:[30]

```
void ldestroy(node *L) {
    node *p;
    while (L) {
        p = L->next;
        free(L);
        L = p;
    }
}
```

Valgrind Output:[31]

```
==10160== HEAP SUMMARY:
==10160==     in use at exit: 0 bytes in 0 blocks
==10160==   total heap usage: 10 allocs, 10 frees, 1,168 bytes allocated
==10160==
==10160== All heap blocks were freed -- no leaks are possible
==10160==
==10160== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Analysis: Perfect! All allocated memory was freed. This is the ideal output.

Example 4: Possibly Lost Memory

Problematic Code:[32]

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

int main() {
    int *p, *q;
    p = (int *)malloc(10 * sizeof(int));
    q = p + 5;  // q points to middle of the block
    p = (int *)malloc(5 * sizeof(int));  // p now points elsewhere!
    exit(0);
}
```

Valgrind Output:[33]

```
==4155== HEAP SUMMARY:
==4155==     in use at exit: 60 bytes in 2 blocks
```

[30] valgrind.pdf
[31] valgrind.pdf
[32] valgrind.pdf
[33] valgrind.pdf

```
==4155==   total heap usage: 2 allocs, 0 frees, 60 bytes allocated
==4155==
==4155== LEAK SUMMARY:
==4155==   definitely lost: 0 bytes in 0 blocks
==4155==   indirectly lost: 0 bytes in 0 blocks
==4155==     possibly lost: 40 bytes in 1 blocks
==4155==    still reachable: 20 bytes in 1 blocks
```

Analysis: The first block (10 ints = 40 bytes) is "possibly lost" because only an interior pointer `q` points to it. Valgrind can't be sure if this is intentional.

Example 5: Array Overflow - Invalid Write

Buggy Code:[34]

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

int main() {
    int n = 16, i, *A;
    A = (int *)malloc(n * sizeof(int));
    printf("A starts at %p, and ends at %p\n", A, A+n-1);

    // Off-by-one error: should be i < n or i = 0 to n-1
    for (i = 1; i <= n; ++i) A[i] = i * i;
    for (i = 1; i <= n; ++i) printf("%d ", A[i]);

    printf("\n");
    free(A);
    exit(0);
}
```

Valgrind Output:[35]

```
==13180== Invalid write of size 4
==13180==    at 0x109240: main (in /home/abhij/.../a.out)
==13180==  Address 0x4a5a080 is 0 bytes after a block of size 64 alloc'd
==13180==    at 0x483B7F3: malloc (in .../vgpreload_memcheck-amd64-linux.so)
==13180==    by 0x1091EC: main (in /home/abhij/.../a.out)
==13180==
==13180== Invalid read of size 4
==13180==    at 0x10926B: main (in /home/abhij/.../a.out)
==13180==  Address 0x4a5a080 is 0 bytes after a block of size 64 alloc'd
```

[34] valgrind.pdf
[35] valgrind.pdf

Analysis: Array indices go from `A` to `A[^14]`, but the loop uses `A[^1]` to `A[^12]`. Writing to `A[^12]` is out of bounds!

Fix:

```
for (i = 0; i < n; ++i) A[i] = (i+1) * (i+1);
for (i = 0; i < n; ++i) printf("%d ", A[i]);
```

Example 6: Buffer Overflow - String Operations

Vulnerable Code:[36]

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

int main(int argc, char *argv[]) {
    char *wnote = malloc(32);
    if (argc == 1) exit(1);

    printf("The input has size %ld\n", strlen(argv[^1]));
    printf("wnote starts at %p and ends at %p\n", wnote, wnote + 31);

    sprintf(wnote, "Welcome to %s", argv[^1]);  // Potential overflow!
    printf("%s\n", wnote);

    free(wnote);
    exit(0);
}
```

Running with long input:

```
valgrind ./a.out "Systems Programming Laboratory"
```

Valgrind Output:[37]

```
The input has size 30
wnote starts at 0x4a5a040 and ends at 0x4a5a05f
==12432== Invalid write of size 1
==12432==    at 0x483EF64: sprintf (in .../vgpreload_memcheck-amd64-linux.so)
==12432==  Address 0x4a5a060 is 0 bytes after a block of size 32 alloc'd
==12432==
==12432== Invalid write of size 1
==12432==  Address 0x4a5a069 is 9 bytes after a block of size 32 alloc'd
```

[36] valgrind.pdf
[37] valgrind.pdf

Analysis: "Welcome to" = 11 chars + "Systems Programming Laboratory" = 30 chars + null terminator = 42 bytes total, but only 32 bytes allocated!

Fix:

```
char *wnote = malloc(strlen(argv[^1]) + 12);  // "Welcome to " + input + '\0'
sprintf(wnote, "Welcome to %s", argv[^1]);
```

Or use safer functions:

```
snprintf(wnote, 32, "Welcome to %s", argv[^1]);  // Prevents overflow
```

Additional Useful Options

Compilation Best Practices

Compile with debugging symbols for better error reports:[38][39]

```
gcc -g -O1 myprogram.c -o myprogram
```

- `-g`: Adds debugging information (line numbers, function names)
- `-O1`: Light optimization that doesn't interfere with debugging

Combining Options

Comprehensive memory check:

```
valgrind --leak-check=full \
         --show-leak-kinds=all \
         --track-origins=yes \
         --log-file=valgrind-report.txt \
         ./myprogram
```

Automated testing:

```
valgrind -q --error-exitcode=1 --leak-check=full ./myprogram
if [ $? -eq 1 ]; then
    echo "Memory errors detected!"
fi
```

Quick Reference Table

Option	Purpose	Example
`--leak-check=full`	Detailed leak information	`valgrind --leak-check=full ./prog`

[38] https://docs.oracle.com/en/operating-systems/oracle-linux/6/porting/ch02s05s02.html
[39] https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html

Option	Purpose	Example
`--track-origins=yes`	Track uninitialized values	`valgrind --track-origins=yes ./prog`
`--log-file=<file>`	Save output to file	`valgrind --log-file=out.txt ./prog`
`-q` or `--quiet`	Show only errors	`valgrind -q ./prog`
`--error-exitcode=N`	Exit code on errors	`valgrind --error-exitcode=99 ./prog`
`--num-callers=N`	Stack trace depth	`valgrind --num-callers=20 ./prog`
`--show-leak-kinds=all`	Show all leak types	`valgrind --show-leak-kinds=all ./prog`

Best Practices

1. **Always compile with `-g`** for meaningful error messages[40][41]
2. **Fix errors in order** - later errors may be caused by earlier ones[42]
3. **Aim for zero errors** - especially "definitely lost" and "invalid" errors[43]
4. **Free all allocated memory** before program exits[44]
5. **Use array bounds carefully** - respect allocated sizes[45]
6. **Be careful with string operations** - check buffer sizes[46]

Common Pitfalls to Avoid

Off-by-one errors:

```
// Wrong: accesses array[n]
for (i = 0; i <= n; i++) array[i] = value;

// Correct: accesses array[^0] to array[n-1]
for (i = 0; i < n; i++) array[i] = value;
```

Losing pointer references:

[40] https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html
[41] https://docs.oracle.com/en/operating-systems/oracle-linux/6/porting/ch02s05s02.html
[42] https://valgrind.org/docs/manual/quick-start.html
[43] valgrind.pdf
[44] valgrind.pdf
[45] valgrind.pdf
[46] valgrind.pdf

```c
// Wrong: original pointer lost
int *p = malloc(100);
p = malloc(200);  // First block leaked!

// Correct: free before reassigning
int *p = malloc(100);
free(p);
p = malloc(200);
```

Interior pointers:

```c
// Risky: only interior pointer remains
int *p = malloc(10 * sizeof(int));
p = p + 5;  // Lost beginning of block!

// Better: keep original pointer
int *p = malloc(10 * sizeof(int));
int *q = p + 5;  // Use q, keep p for free()
```

Summary

Valgrind is an essential tool for C/C++ programmers to detect and fix memory errors. It helps identify:[47]

- **Memory leaks** (allocated but not freed memory)
- **Invalid memory access** (reading/writing outside allocated bounds)
- **Uninitialized value usage**
- **Double-free errors**

By running your programs through Valgrind and fixing all reported errors, you can write more robust, reliable, and correct code. The goal is to achieve the perfect output: **"All heap blocks were freed – no leaks are possible"** with **"0 errors from 0 contexts"**.[48]

Start simple with `valgrind ./myprogram`, then add options like `--leak-check=full` as needed. Always remember: **a Valgrind-clean program is a well-written program**.[49]

[47] valgrind.pdf
[48] valgrind.pdf
[49] https://valgrind.org/docs/manual/quick-start.html