Example *valgrind* run on an Asterix compiler with memory errors

G.P. Halkes

November 9, 2006

Introduction

valgrind is a tool with which memory access errors and memory leaks can be detected. This document shows examples of the most common errors in the Compiler Construction labwork.

Memory leaks

Below is the output from *valgrind* on a version of the Asterix compiler into which deliberate memory leaks have been introduced. The compiler was executed by:

```
valgrind --tool=memcheck --leak-check=full --show-reachable=yes ./cbf
                                -I../include ../example/example.ast
```

```
==24947== Memcheck, a memory error detector for x86-linux.
==24947== Copyright (C) 2002-2005, and GNU GPL’d, by Julian Seward et al.
==24947== Using valgrind-2.4.0, a program supervision framework for x86-linux.
==24947== Copyright (C) 2000-2005, and GNU GPL’d, by Julian Seward et al.
==24947== For more details, rerun with: -v
==24947==
==24947==
==24947== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
==24947== malloc/free: in use at exit: 653 bytes in 45 blocks.
==24947== malloc/free: 914 allocs, 869 frees, 53208 bytes allocated.
==24947== For counts of detected errors, rerun with: -v
==24947== searching for pointers to 45 not-freed blocks.
==24947== checked 78236 bytes.
==24947==
==24947== 224 bytes in 14 blocks are definitely lost in loss record 1 of 2
==24947==    at 0x1B90459D: malloc (vg_replace_malloc.c:130)
==24947==    by 0x804AB12: safe_malloc (salloc.c:22)
==24947==    by 0x804BE66: new_subprog_type (type.c:47)
==24947==    by 0x804C67A: yyparse (grammar.y:240)
==24947==    by 0x804AAA7: main (main.c:85)
==24947==
==24947==
==24947== 429 bytes in 31 blocks are still reachable in loss record 2 of 2
==24947==    at 0x1B90459D: malloc (vg_replace_malloc.c:130)
==24947==    by 0x804AB12: safe_malloc (salloc.c:22)
==24947==    by 0x804AC89: new_scope (scope.c:77)
==24947==    by 0x804C4B6: yyparse (grammar.y:127)
==24947==    by 0x804AAA7: main (main.c:85)
==24947==
==24947== LEAK SUMMARY:
```

```
31 ==24947==    definitely lost: 224 bytes in 14 blocks.
32 ==24947==      possibly lost: 0 bytes in 0 blocks.
33 ==24947==    still reachable: 429 bytes in 31 blocks.
34 ==24947==         suppressed: 0 bytes in 0 blocks.
```

On line 15 of the output, *valgrind* mentions that there are 14 blocks definitely lost. This is caused by failing to call `delete_type` at declaration.c:52. Note that the compiler was specifically altered not to call `delete_type` for this run.

Line 23 of the *valgrind* output notes that there are 31 blocks still reachable. Again, this is caused by deliberatly disabling a line of code in the compiler, in this case main.c:64 (`delete_scope`). The difference with the previous message is that the pointers to the blocks allocated for the first message are lost, while the program still has a usable pointer to the blocks in the second case. Therefore, *valgrind* says the blocks are definitely lost in the first case, and still reachable in the second case.

Following these lines is a stacktrace of the points where `malloc` was called for the blocks that where not `free`'d. In the Asterix compiler, all calls to `malloc` are done through the `safe_malloc` function which ensures that failing calls are caught. When going down through the stacktrace, it is easy to see that both calls to `malloc` originated in the grammar (function `yyparse` in grammar.y).

Freeing a block twice or reading from `free`'d memory

If the compiler calls one of the `delete_XXX` functions on the same block of memory more than once, the compiler may seem to operate correctly. However, on a different platform, the compiler may suddenly exhibit broken behaviour. Therefore you should check whether your compiler can run through valgrind without errors. Below is part of the *valgrind* output for an Asterix compiler with a trivial double call to the `delete_type` function:

```
==25102== Invalid read of size 4
==25102==    at 0x804BECE: delete_type (type.c:68)
==25102==    by 0x8048B07: delete_declaration (declaration.c:54)
==25102==    by 0x8048B71: delete_declaration_list (declaration.c:75)
==25102==    by 0x804AA6F: terminate (main.c:63)
==25102==    by 0x804AACF: main (main.c:102)
==25102==  Address 0x1BA4BF14 is 4 bytes inside a block of size 16 free'd
==25102==    at 0x1B904B04: free (vg_replace_malloc.c:152)
==25102==    by 0x804BEFB: delete_type (type.c:80)
==25102==    by 0x8048AF9: delete_declaration (declaration.c:53)
==25102==    by 0x8048B71: delete_declaration_list (declaration.c:75)
==25102==    by 0x804AA6F: terminate (main.c:63)
==25102==    by 0x804AACF: main (main.c:102)
```

As you can see, *valgrind* reports an invalid read. This is caused by the `delete_type` function checking members of the struct. The first stacktrace shows where the invalid read is performed, the second stacktrace shows where `free` was called to release the block of memory. If you study the stacktrace closely you will find that `delete_type` is called from declaration.c:53 and declaration.c:54. In this case the compiler contained a duplicate of line 53 at line 54. This example does not have a so-called *double free* error, because the code in the `delete_type` function does reference counting.

Double free

A double free error is when `free` is called with the same pointer more than once. This can occur for example when a `delete_XXX` function is called for anything but a `Type` struct. The associated output from *valgrind* looks like:

```
==25841== Invalid free() / delete / delete[]
==25841==    at 0x1B904B04: free (vg_replace_malloc.c:152)
==25841==    by 0x804BDA9: delete_token (token.c:28)
==25841==    by 0x80493A8: delete_expression (expression.c:49)
==25841==    by 0x804B1EF: delete_statement (statement.c:175)
```

```
==25841==    by 0x804B133: delete_statement (statement.c:148)
==25841==    by 0x8048ADD: delete_declaration (declaration.c:50)
==25841==    by 0x8048B71: delete_declaration_list (declaration.c:75)
==25841==    by 0x804AA6F: terminate (main.c:63)
==25841==    by 0x804AACF: main (main.c:102)
==25841==  Address 0x1BA50138 is 0 bytes inside a block of size 23 free'd
==25841==    at 0x1B904B04: free (vg_replace_malloc.c:152)
==25841==    by 0x804BDA9: delete_token (token.c:28)
==25841==    by 0x80493A8: delete_expression (expression.c:49)
==25841==    by 0x804B1EF: delete_statement (statement.c:175)
==25841==    by 0x804B133: delete_statement (statement.c:148)
==25841==    by 0x8048ACF: delete_declaration (declaration.c:49)
==25841==    by 0x8048B71: delete_declaration_list (declaration.c:75)
==25841==    by 0x804AA6F: terminate (main.c:63)
==25841==    by 0x804AACF: main (main.c:102)
```

To create this trace, the `delete_statement` was called twice from `delete_declaration`. Once from line 49, and once from line 50.

Final remarks

Some versions of glibc and flex do not clean up properly after finishing. Specifically, you may see blocks allocated in the `fopen` and `yylex` functions. If the original compiler exhibits the same behaviour, you do not need to fix this. Also, when an error is detected by the Bison generated parser, some blocks allocated from `yyparse` will not be `free'd`. For the Bison compiler it is impossible to avoid this problem in a way that will work with all Bison versions. Therefore this is acceptable behaviour.