Course
ELEC-A7151
Course materials
Your points
Code
Code Vault
Course Pages
MyCourses
Teams Channel

This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

« 3 Git Course materials
ELEC-A7151 / Software development tools / 4 Valgrind

4 Valgrind¶

Contents

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. In this course, we will be using Memcheck tool, which is a memory error detector.

Valgrind can be run in Linux command line with the command

```
valgrind ./main.out
```

where `./main.out` is the name of your executable produced when compiling your code.

Caution

Valgrind might not work properly on macOS. It might show memory leaks when there are none etc. Be cautious if you try valgrind on a Mac.

Note

Valgrind is not available for Windows (but it is for WSL, of course).

The Valgrind Memcheck tool has several command line options, but we will be using the following three options.

- `--leak-check=full` : Memcheck shows each individual leak in detail.
- `--track-origins=yes` : Memcheck keeps track of the origins of all uninitialised values. Then, when an uninitialised value error is reported, Memcheck will try to show the origin of the value. It might also report their line numbers if is the program was compiled with the `-g` flag of `gcc` compiler.
- `--show-leak-kinds=all` : Memcheck shows all leak kinds in its full leak search.

Therefore, you can run Valgrind as follows:

```
valgrind --leak-check=full --track-origins=yes --show-leak-kinds=all ./main.out
```

Hint

Simple heuristics for reading the valgrind outputs:

```
int a = 0;
char b = 0;
char* c = "test";
printf("%zd %zd %zd\n", sizeof(a), sizeof(b), sizeof(c));
```

The example prints `4 1 8`, so based on the size the valgrind reports, you can figure out the type of the value being reported. Note that the variable `c` has a size of 8, because it is a pointer. All pointers are equal in size regardless what is the type of value they hold inside.

Som example of valgrind outputs are given below.

4.1 Invalid write of size 1¶

```
==360== Invalid write of size 1
==360==    at 0x4C2588C: strcpy (mc_replace_strmem.c:311)
==360==    by 0x402EE1: init_product (products.c:22)
==360==    by 0x4018F0: test_init_product (test_source.c:75)
==360==    by 0x405E10: srunner_run_all (in /tmc/test/test)
==360==    by 0x402AEA: tmc_run_tests (tmc-check.c:122)
==360==    by 0x4024D6: main (test_source.c:261)
==360==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

or

```
==360== Invalid write of size 1
==360==    at 0x4C2588C: strcpy (mc_replace_strmem.c:311)
==360==    by 0x402EF5: init_product (products.c:22)
==360==    by 0x4018F0: test_init_product (test_source.c:75)
==360==    by 0x405E10: srunner_run_all (in /tmc/test/test)
==360==    by 0x402AEA: tmc_run_tests (tmc-check.c:122)
==360==    by 0x4024D6: main (test_source.c:261)
==360==  Address 0x518d0d8 is 0 bytes after a block of size 8 alloc'd
==360==    at 0x4C244E8: malloc (vg_replace_malloc.c:236)
==360==    by 0x402ED5: init_product (products.c:21)
==360==    by 0x4018F0: test_init_product (test_source.c:75)
==360==    by 0x405E10: srunner_run_all (in /tmc/test/test)
==360==    by 0x402AEA: tmc_run_tests (tmc-check.c:122)
==360==    by 0x4024D6: main (test_source.c:261)
```

The memory that was written to, has not been allocated properly. Before using `strcpy` , the destination needs to have sufficient space reserved.

- In the first case memory has not been allocated at all: `Address 0x0 is not stack'd, malloc'd or (recently) free'd`
  - `0x0` refers to a `NULL` pointer
  - `not stack'd, malloc'd or (recently) free'd` refers to the fact that the pointer being used has not been reserved from the stack or heap or been free'd recently
- The second case has too little memory reserved for the string: `Address 0x518dd08 is 0 bytes after a block of size 8 alloc'd`
  - `0 bytes after a block of size 8 alloc'd` tells that we have allocated 8 bytes, and that we're trying to write one byte after the reserved memory

4.2 Invalid write of size 4¶

```
==359== Invalid write of size 4
==359==    at 0x4023E1: add_to_array (source.c:45)
==359==    by 0x4013A2: test_add_to_array (test_source.c:55)
==359==    by 0x405210: srunner_run_all (in /tmc/test/test)
==359==    by 0x401EE6: tmc_run_tests (tmc-check.c:122)
==359==    by 0x4018FF: main (test_source.c:82)
==359==  Address 0x518d428 is 24 bytes inside a block of size 25 alloc'd
==359==    at 0x4C2455E2: realloc (vg_replace_malloc.c:525)
==359==    by 0x40234E: add_to_array (source.c:36)
==359==    by 0x4013A2: test_add_to_array (test_source.c:55)
==359==    by 0x405210: srunner_run_all (in /tmc/test/test)
==359==    by 0x401EE6: tmc_run_tests (tmc-check.c:122)
==359==    by 0x4018FF: main (test_source.c:82)
```

The allocated memory is too little for the operation we're trying to perform because a value is being written past the end of the allocated memory. Usually this happens due to a math error.

incorrect: `malloc(arr,(n*sizeof(int)+1))`

Here the calculations result in reserving space for `n` integers and the `+1` at the end only add one more byte of space, which is not enough for an integer.

correct: `malloc(arr,((n+1)*sizeof(int))`

4.3 Invalid read of size 4¶

```
==360== Invalid read of size 4
==360==    at 0x402680: getDenom (fraction.c:65)
==360==    by 0x401944: printFr (test_source.c:77)
==360==    by 0x401A76: test_add (test_source.c:94)
==360==    by 0x405580: srunner_run_all (in /tmc/test/test)
==360==    by 0x4021A6: tmc_run_tests (tmc-check.c:122)
==360==    by 0x401E61: main (test_source.c:159)
==360==  Address 0x3 is not stack'd, malloc'd or (recently) free'd
```

The address we're trying to read from (by dereferencing a pointer or indexing an array), has not been allocated at all. This usually happens when you create a variable but do not set it's value at all.

4.4 Invalid free() / delete / delete[]¶

```
==359== Invalid free() / delete / delete[]
==359==    at 0x4C240FD: free (vg_replace_malloc.c:366)
==359==    by 0x4022E6: add_to_array (source.c:41)
==359==    by 0x4019B2: test_add_to_array (test_source.c:55)
==359==    by 0x4051A0: srunner_run_all (in /tmc/test/test)
==359==    by 0x401E76: tmc_run_tests (tmc-check.c:122)
==359==    by 0x4018F2F: main (test_source.c:82)
==359==    Address 0x518d3b0 is 0 bytes inside a block of size 36 free'd
==359==    at 0x4C2455E2: realloc (vg_replace_malloc.c:525)
==359==    by 0x4022C5: add_to_array (source.c:38)
==359==    by 0x4019B2: test_add_to_array (test_source.c:55)
==359==    by 0x4051A0: srunner_run_all (in /tmc/test/test)
==359==    by 0x401E76: tmc_run_tests (tmc-check.c:122)
==359==    by 0x4018F2F: main (test_source.c:82)
```

This happens when trying to free something that has already been free'd. For example with realloc `new_arr = realloc(arr, (num+1)*sizeof(int));` the `arr` is being free'd automatically and trying to run `free(arr);` after this realloc will cause an error seen above. Note the line `Address 0x518d3b0 is 0 bytes inside a block of size 36 free'd`, which describes the situation.

```
==364== Invalid free() / delete / delete[]
==364==    at 0x4C240FD: free (vg_replace_malloc.c:366)
==364==    by 0x4013BC: delete_product (products.c:95)
==364==    by 0x402235: test_delete_product (test_source.c:219)
==364==    by 0x406100: srunner_run_all (in /tmc/test/test)
==364==    by 0x4020A1: tmc_run_tests (tmc-check.c:122)
==364==    by 0x402509: main (test_source.c:265)
==364== Address 0x51900A0 is 64 bytes inside a block of size 128 alloc'd
==364==    at 0x4C245E2: realloc (vg_replace_malloc.c:525)
==364==    by 0x402FD0: add_product (products.c:40)
==364==    by 0x4021E5: test_delete_product (test_source.c:212)
==364==    by 0x406100: srunner_run_all (in /tmc/test/test)
==364==    by 0x4020A1: tmc_run_tests (tmc-check.c:122)
==364==    by 0x402509: main (test_source.c:265)
```

This problem is most probably caused by trying to free a part of reserved memory. After reserving an array of consecutive places for values, it is not possible to free the space for just one value, but instead to remove the value you need to move the other values so that the empty space is at the end of the reserved buffer and after that the space can be reduced with realloc.

This error also happens, if you're trying to free a pointer, which doesn't point to the beginning of the reserved space. This example would result in a similar error:

```
int* array = malloc(10*sizeof(int));
array++;
free(array);
```

Conditional jump or move depends on uninitialised value(s)

```
==26196== Conditional jump or move depends on uninitialised value(s)
==26196==    at 0x4C2CB94: strcmp (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26196==    by 0x40183B: test_product (test_source.c:40)
==26196==    by 0x4018A9: test_init_product (test_source.c:71)
==26196==    by 0x406448: srunner_run_all (in /home/home1/13/jukkale1/unix/CC/Module_3/Task_3/test/test)
==26196==    by 0x402D94: tmc_run_tests (tmc-check.c:122)
==26196==    by 0x402739: main (test_source.c:254)
```

Here something has been left uninitialised: strcmp is trying to compare with a value that is uninitialised. Maybe the string has more space reserved than there are characters and the null terminator is missing or in the wrong place? Or maybe the conditional clause contains a value that is uninitialised? For example:

```
char str[10];
char str2[]="hello!";
strncpy(str, str2, 6);
str[9]='\0';
```

here we are left with some uninitialised values in `str`. This could be fixed by setting the null terminator to the correct position or by using `memset` before `strncpy`.

```
==360== Conditional jump or move depends on uninitialised value(s)
==360==    at 0x4017FF: release_list (test_source.c:53)
==360==    by 0x40196E: test_add_product (test_source.c:81)
==360==    by 0x406150: srunner_run_all (in /tmc/test/test)
==360==    by 0x402CD2: tmc_run_tests (tmc-check.c:122)
==360==    by 0x4026BE: main (test_source.c:306)
==360==  Unintitialised value was created by a heap allocation
==360==    at 0x4C244E8: malloc (vg_replace_malloc.c:236)
==360==    by 0x4030DE: add_product (list.c:20)
==360==    by 0x40195E: test_add_product (test_source.c:80)
==360==    by 0x406150: srunner_run_all (in /tmc/test/test)
==360==    by 0x402CD2: tmc_run_tests (tmc-check.c:122)
==360==    by 0x4026BE: main (test_source.c:306)
```

Here the uninitialised value is due to a heap allocation, where something dynamically allocated has not been initialised. For example, when reserving space for a struct, if not all of its members are initialised, we are left with an uninitialised value.

4.5 N bytes in N blocks are definitely lost in loss record NN of NN¶

```
==361== 60 bytes in 4 blocks are definitely lost in loss record 91 of 94
==361==    at 0x4C244E8: malloc (vg_replace_malloc.c:236)
==361==    by 0x402EF7: init_product (products.c:16)
==361==    by 0x402FD2: add_product (products.c:37)
==361==    by 0x401AED: test_add_product (test_source.c:102)
==361==    by 0x405F80: srunner_run_all (in /tmc/test/test)
==361==    by 0x4020A1: tmc_run_tests (tmc-check.c:122)
==361==    by 0x402509: main (test_source.c:265)
```

Memory has been lost/leaked, which means that a pointer to a reserved memory has been lost and the memory has not been released. Check that for every `malloc (calloc)` or `new` you have a corresponding `free` or `delete`.