Memory Debugging Tutorial

Presented by Eta Kappa Nu

Download the exercises

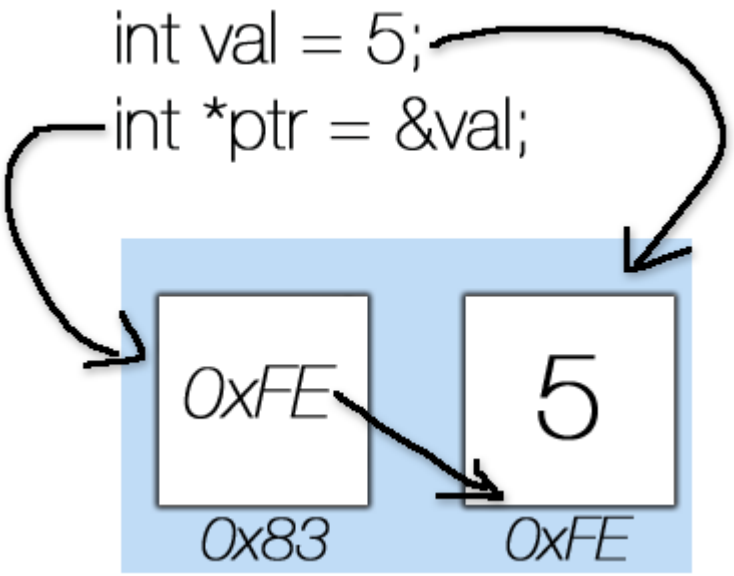
Run on your terminal:

```
git clone git://github.com/HKNTutorials/memory-intro.git
```

Once you've download the code, you can compile all the examples by running "make". (And the same thing to recompile).

Pointers

- Pointers are variables whose values are memory addresses
- Often, we represent them visually as arrows.
- Notation:
  - `int*` is a pointer to an `int`
  - & means "the address of"
  - `*x` means dereference x (load or store to address given by x)



images from xkcd.com/138/ and technology.stepbystep.com/how-to-access-memory-in-c-using-pointers-146/

Dynamic Memory Allocation

- `malloc(x)` allocates a chunk of memory `x` bytes long, and returns a pointer to it
- `free(ptr)` deallocates memory (ptr should have been the return value of some `malloc` call)
- `free` should be called exactly once per `malloc`'ed chunk of memory.
- `free'ing` a `NULL` pointer is valid. `free'ing` a pointer twice, or using a pointer after it is `free'd`, yields undefined behavior.

Simple Pointer Example

```
typedef struct {
    char *name;
    unsigned int age;
} person;

typedef struct {
    int width, height;
    unsigned int* data;
    person* photographer;
} image;
```

```
int main() {
    image* img = (image*)
        malloc(sizeof(image));
    fill_image(img, 5, 5);
    person* p = (person*)
        malloc(sizeof(person));
    fill_person(p);
    img->photographer = p;
}
```

GDB It!

```
$ gdb simpleexample
(gdb) start
...
Breakpoint 1, main () at simpleexample.c:39
39    image* img = (image*) malloc(sizeof
(image));
(gdb) break 44
Breakpoint 2 at 0x100000ec1: file simpleexample.
c, line 44.
(gdb) continue
Continuing.
```

GDB It!

```
Breakpoint 2, main () at simpleexample.c:44
44    return 0;
(gdb) print img
$1 = (image *) 0x100100980
(gdb) print *img
$2 = {
  width = 5,
  height = 5,
  data = 0x1001009a0,
  photographer = 0x1001009c0
}
```

GDB It!

```
(gdb) print img->photographer
$3 = (person *) 0x1001009c0
(gdb) print *img->photographer
$4 = {
  name = 0x100000ef8 "Ben Bitdiddle",
  age = 20
}
(gdb) x/25wd img->data
0x1001009a0:  0  1  2  3
0x1001009b0:  4  1  2  3
0x1001009c0:  4  5
```

Valgrind

- Tool for memory debugging
- Command is often "`valgrind --leak-check=full <program>`"
- Finds bugs relating to dynamically allocated memory (invalid reads, writes, frees, etc.)

A few extra useful flags:

--show-reachable=yes - use to show which objects are still reachable at the end of the program

--track-origins=yes - use to track uninitialized value - will tell you where the uninitialized value is created. (it's more informative for uninitialized heap than for uninitialized stack)

Valgrind output for simpleexample.c

valgrind --leak-check=full ./simpleexample
==17550== Memcheck, a memory error detector
==17550== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==17550== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==17550== Command: ./simpleexample
==17550==
==17550== HEAP SUMMARY:
==17550== in use at exit: 140 bytes in 3 blocks
==17550== total heap usage: 3 allocs, 0 frees, 140 bytes allocated
==17550==
==17550== **140 (24 direct, 116 indirect) bytes** in 1 blocks are definitely lost in loss record 3 of 3
==17550==at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==17550==by 0x4005DA: main (**simpleexample.c:39**) <-- what was allocated here?
==17550==

Valgrind output (continued)

```
==17550== LEAK SUMMARY:
==17550==definitely lost: 24 bytes in 1 blocks
==17550==indirectly lost: 116 bytes in 2 blocks
==17550==         possibly lost: 0 bytes in 0 blocks
==17550==still reachable: 0 bytes in 0 blocks
==17550==         suppressed: 0 bytes in 0 blocks
==17550==
==17550== For counts of detected and suppressed errors, rerun with: -v
==17550== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Common Mistakes

- free'ing a malloc'ed pointer two or more times
- using a pointer after it's been free'd
- forgetting to free a malloc'ed pointer (memory leak)
- out of bounds array access (for dynamically allocated arrays, out of bounds writes often corrupt the heap - may cause malloc & free to later crash)

Exercises: Overview

- copy.c - makes objects, prints out some info, then destroys them
- array_manipulation.c - dynamic array manipulation
- reverselist.c - reverses a linked list
- global.c - a short instructive example to show what valgrind can and can't detect. The comments should explain it.
- wc.c - a program that counts words

copy.c

This program allocates, prints out, copies, and frees some objects. However, it mismanages its memory... can you figure it out?

array_manipulation.c

This program has a dynamically allocated array, which it grows and shrinks on demand. It initially has length 10 and asks for 10 integers; when you give it a new size, it allocates new space for the array, and either truncates the array to fit in the new space or asks you for integers for the new slots of the array.

Of course, this program is buggy. Hint: valgrind is super helpful.

reverse_list.c

...
==700== Command: reverselist
==700==
==700== Invalid **write of size 8**
==700== at 0x100000BFB: make_linked_list (reverselist.c:28)
==700== by 0x100000DB5: main (reverselist.c:113)
==700== Address 0x100004190 is 0 bytes **inside a block of size 4 alloc'd**
==700== at 0xE0D6: malloc (vg_replace_malloc.c:274)
==700== by 0x100000BD4: **make_linked_list (reverselist.c:22)**
==700== by 0x100000DB5: main (reverselist.c:113)
==700== Invalid read of size 8
...

reverse_list.c output

original:
Element #0: 3
Element #1: 56
Element #2: 2341
Element #3: 90
Element #4: 275
Element #5: -24
Element #6: 32
Element #7: 64
Element #8: 77

reversed:
Element #0: 77
Element #1: 64
Element #2: 32
Element #3: -24
Element #4: 275
Element #5: 90
Element #6: 2341
Element #7: 56

Check number of elements and compare against code. Don't trust the printing!

reverse_list valgrind output

==700== HEAP SUMMARY:

...

==700== **8 bytes in 2 blocks are definitely lost** in loss record 2 of 10

==700== at 0xE0D6: malloc (vg_replace_malloc.c:274)

==700== by 0x100000BF3: **make_linked_list (reverselist.c:28)** <-- allocation of lost memory

==700== by 0x100000DB5: main (reverselist.c:113)

==700==

==700== LEAK SUMMARY:

==700== definitely lost: 8 bytes in 2 blocks

==700== indirectly lost: 0 bytes in 0 blocks

==700== possibly lost: 0 bytes in 0 blocks

==700== still reachable: 18,579 bytes in 33 blocks

==700== suppressed: 0 bytes in 0 blocks

...

Word Count (wc.c)

This examples counts seem rather inconsistent, so it makes sense to step through the code. Having said that, the issues are more conceptual than mistakes with memory management.

It might be helpful to display the value of `words` and the character's place in the string (`*str`).

```
(gdb) display str
```