

# M7 – Memory Checkers

CS 136L S23 – Week 10

Yiqing Irene Huang



UNIVERSITY OF  
**WATERLOO**

FACULTY OF MATHEMATICS  
DAVID R. CHERITON SCHOOL  
OF COMPUTER SCIENCE

# Disclaimer

- The following slides were not presented page by page in class.
- They are my own study notes to share with students.
- In the lab session, we will cover key points, do small demos and give hints on commonly seen errors

# Main Points

Use memory checkers to identify memory errors.

- Valgrind directly operates on the executable
  - Supports both clang and gcc compiled code, works better with gcc
  - memcheck tool
- AddressSanitizer injects instrumentation at compile time
  - Part of clang, gcc version 4.8 and up also supports it.

# Memory Errors

1. Using uninitialized memory
2. Dereferencing/accessing a NULL pointer or an invalid address.
3. Buffer overflow
4. Using stack memory after function has returned
5. Memory Leak
6. Accessing dynamic memory beyond the range of memory allocated
7. Accessing dynamic memory that has already been deallocated
8. Trying to deallocate memory that is no-longer/not ours to deallocate

# Lab Thresholds

| Question | Description           | # of Tests | Pass | Complete |
|----------|-----------------------|------------|------|----------|
| Q1       | Allocating a node     | 4          | 1    | 3        |
| Q2       | Deallocating a node   | 4          | 1    | 3        |
| Q3       | Printing a list       | 4          | 1    | 3        |
| Q4       | Computing list length | 4          | 1    | 3        |

## Valgrind

```
gcc -Wall -O0 -g <files-in-the-program-come-here>  
valgrind -s --leak-check=full --track-origins=yes ./executable-name
```

## AddressSanitizer

```
clang -Wall -O0 -g -fsanitize=address <files-in-the-program-come-here>  
./executable-name
```

## Bug Reporter

```
/u2/cs1361/pub/lab7/bug-reporter
```

How to name the executable?  
What is a.out?

# Introduction

- Valgrind

```
1 gcc -Wall -g -O0 test.c -o test
2 valgrind ./test
```

```
1 // File test.c
2 #include <stdio.h>
3
4 int main(void) {
5     printf("Hello World!\n");
6     return 0;
7 }
```

- AddressSanitizer

```
1 clang -O1 -fsanitize=address -fno-omit-frame-pointer -Wall -g test.c -o test
2 ./test
```

Which optimization level does the module recommend to use when using Address Sanitizer?

# Uninitialized Memory

- The `-Wall` option is required to identify the error both for clang and gcc at compile time
- Clang: `-fsanitize=address` at compile time
- Valgrind will capture the memory error at runtime.
  - Use `-track-origins=yes` option for details
- AddressSanitizer will not report runtime error
  - `-O0` gives consistent output
  - `-O1` gives random output

```
1 // File test1.c
2 #include <stdio.h>
3
4 int main(void){
5     int i;
6     printf("%d",i);
7     return 0;
8 }
```

# Conditional Jump Over Uninitialized Values

- The `-Wall` option is required to identify the error for gcc
- clang cannot capture this error at compile time
- Valgrind will capture the memory error at runtime.
  - Use `-track-origins=yes` option for details
- AddressSanitizer will not report runtime error
- MemorySanitizer will report runtime error

`-fsanitize=memory`

```
1 // File test2.c
2 int main(int argc, char **argv) {
3     int arr[2];
4     if (arr[argc] != 1)
5         return 1;
6     else
7         return 0;
8 }
```

# NULL Address

## Invalid Write

```
1 // File test3.c
2 #include <stdio.h>
3
4 int main(void){
5     int *p = NULL;
6     *p = 5;
7     printf("%d\n", *p);
8 }
```

Segmentation fault both for clang and gcc

## Invalid Read

```
1 // File test4.c
2 #include <stdio.h>
3
4 int main(void){
5     int *p = NULL;
6     int x = *p;
7     printf("%d\n", x);
8 }
```

Segmentation fault for gcc  
Only SEGV with -O0 for clang

# Buffer Overflow

```
1 //File: buffer1.c
2 #include <stdio.h>
3
4 void print_element(int *arr, int index){
5     printf("%d",arr[index]);
6 }
7 int main(void) {
8     int a[10];
9     print_element(a,10);
10 }
```

```
1 #include <stdio.h>
2 int main(void){
3     int a = 5;
4     printf("%d\n",*(&a + 1));
5 }
```

- Valgrind cannot detect buffer overflow
- AddressSanitizer can detect it with `-O0` only

# Stack Use After Run

- Valgrind cannot detect the error.
- AddressSanitizer can detect the error with special flags

```
1 // File stack0.c
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 char *get_hello(){
6     char str[20] = "Hello World!";
7     char *toRet = str;
8     return toRet;
9 }
10 int main (){
11     printf("%s\n",get_hello());
12     return 0;
13 }
```

```
1 clang -fsanitize=address -Wall -g stack0.c -o stack0
2 ./stack0
3 ??}9EV
4 ./stack0
5 ?!?:V
6
7 ASAN_OPTIONS=detect_stack_use_after_return=1 ./stack0
8 =====
9 ==5169==ERROR: AddressSanitizer: stack-use-after-return on
10 --- output truncated
```

# Stack Use After Run – Cont'd

```
1 // File stack0.c
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 char *get_hello(){
6     char str[20] = "Hello World!";
7     return &str; ←
8 }
9 int main (){
10     printf("%s\n",get_hello());
11     return 0;
12 }
```

```
1 // File stack0.c
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 char *get_hello(){
6     char str[20] = "Hello World!";
7     char *toRet = str; ←
8     return toRet; ←
9 }
10 int main (){
11     printf("%s\n",get_hello());
12     return 0;
13 }
```

- The `-Wall` will detect it both for gcc and clang at compile time

```

1 //file stack1.c
2 #include <stdio.h>
3
4 struct vec {
5     int x;
6     int y;
7 };
8
9 // get_vec(_x,_y) is a helper for creating a
10 //   vec object with the specified values
11 struct vec *get_vec(int _x, int _y){
12     struct vec myVec;
13     struct vec *p = &myVec;
14     p->x = _x;
15     p->y = _y;
16     return p;
17 }
18
19 int main(void){
20     struct vec *p1 = get_vec(1,2);
21     struct vec *p2 = get_vec(5,10);
22
23     printf("%d,%d\n", p1->x, p1->y);
24     printf("%d,%d\n", p2->x, p2->y);
25
26     return 0;
27 }

```

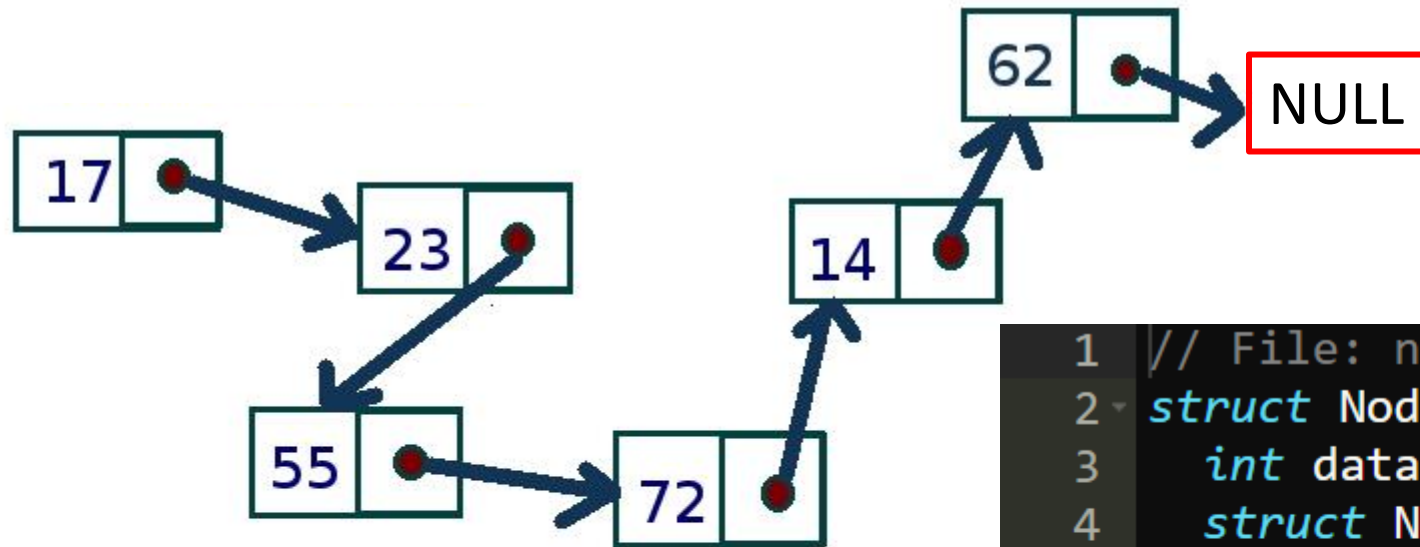
## Stack Use After Run – Cont'd

- Both valgrind and AddressSanitizer will catch the error

# Heap Memory Errors

- Memory leak
  - malloc without the matching free()
- Incorrect use of malloced memory
  - Access memory that is not allocated. Allocate X bytes, access X+n ( $n > 0$ ) bytes
- Incorrect free
  - Free a pointer that is not returned by malloc
- Premature free
  - After free, access the memory

# Linked Structure



```
1 // File: node.h
2 struct Node {
3     int data;
4     struct Node *next;
5 };
6
7 struct Node *list_creator(int length);
8 void list_printer(struct Node *list);
9
```

# Acknowledgement

- Slides by courtesy of Carmen Bruni and Dave Tompkins
- Demo notes from Nomair Naeem
- Demo lectures by Carmen Bruni, Dave Tompkins, and Nomair Naeem

# References

- CS 136L edX notes at <https://online.cs.uwaterloo.ca/>