Arrays, Pointers and Valgrind

Spring 2026 - COMP 201 Lab 3

KOÇ UNIVERSITY

Pointer Recap

- What's a pointer?
  - A variable that stores a memory address of another variable
- Why do we use it?
  - Pointers provide a way to directly interact with memory, allowing a more flexible and efficient data manipulation
- Declaration:
  - A pointer is declared by specifying its data type and name, with an asterisk (*) before the name. **Syntax: data_type *pointer_name;**
  - The data type indicates the type of variable the pointer can point to. For example, "int *ptr;" declares a pointer to an integer.

Pointer Recap

Key operations:

- & (address-of): To get the address of a variable
- * (dereference): To access the value at an address

Example:

```
int x = 7;
int *ptr = &x; //Pointer to x.
/* Dereference pointer to get value of x.
Outputs: "Value of x: 7" */
printf("Value of x: %d\n", *ptr);
/* Print address of x.
Outputs: "Address of x: 0x16efc31b8"
This address will vary each run. */
printf("Address of x: %p\n", (void*)ptr);
```

Pointers and Arrays

Arrays and pointers are closely related because each array name is a pointer to its first element.

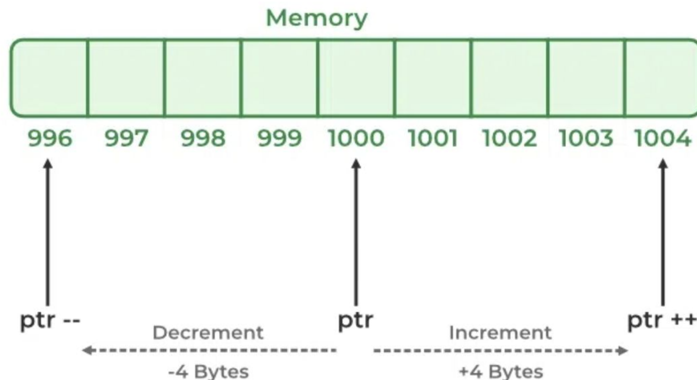
Arr[i] is equal to *(arr+i)

Example:

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;  // arr is same as &arr[0]

printf("%d\n", arr[2]);  // Using array indexing
printf("%d\n", *(arr + 2)); // Using array name arithmetic
printf("%d\n", *(ptr + 2)); // Using pointer arithmetic

//all three print 30
```

Pointer Arithmetics

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. We can use pointer arithmetic to move through memory, when we increment a pointer it points to the next element (**be careful not next byte next element**)



Pointer Arithmetics

Some Operations:

- ptr++ : Move to next element
- ptr + n : Move n elements forward
- ptr1 - ptr2 : Number of elements between pointers

Example: (See when added 2, it moves 8 bytes (2* sizeof(int)))

```c
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;  // arr is same as &arr[0]

printf("%p\n", (&arr[0]));    // address of first element
printf("%p\n", ptr);         // address of first element
printf("%p\n", (ptr+2));     // address of third element
printf("%d\n", *(ptr+2));    // value of third element
printf("%d\n", *(arr+2));    // value of third element
```

```c
/*
outputs something like:
0x16fc771a0
0x16fc771a0
0x16fc771a8
30
30
*/
```

Array Traversal with Pointers

Both methods are equivalent

```c
//Method 1: Array Indexing
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
```

```c
//Method 2: Pointer Arithmetic
for (int i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i));
}
```

2D Arrays

We can represent a 2D array as a one large contiguous block

```
int *array_2d;
// a single NxM malloc: really this is a large 1-dim array of int values onto
which we will map 2D accesses
array_2d = malloc(sizeof(int) * N * M);
// To access element at row i, column j:
int i = 2, j = 3, value = 42;
array_2d[i * M + j] = value;
// Or using pointer arithmetic:
*(array_2d + i * M + j) = value;

// in memory:
//                    row 0       row 1     row 2 ...
// 2d_array -----> [ 0, 0, ... , 0, 0, ... 0, 0, ... ]
```

2D Arrays: Array of Arrays

We can also represent an 2D array as array of arrays. Allocate array of pointers, then allocate each row separately.

```
int **array_2d = malloc(sizeof(int*) * N);  // Array of pointers
for (int i = 0; i < N; i++) {
    array_2d[i] = malloc(sizeof(int) * M);  // Each row
}
int i = 1, j = 2, value = 42;
// in memory:
// **  -----> | *-|-------> [ 0, 0, 0, ... , 0]  row 0
//            | *-|-------> [ 0, 0, 0, ... , 0]  row 1
//            |...|                               ...
//            | *-|-------> [ 0, 0, 0, ... , 0]  row N

// access using [] notation:
// 2d_array[i] is ith bucket in 2d_array, which is the address of
// a 1d array, on which you can use indexing to access its bucket value
// Access is natural:
array_2d[i][j] = value;
```

Freeing 2D Arrays

To free a 2D array we should free the parts in reverse order.

```
// For a single 1D array this works:
free(array_1d);

//But for 2D array, need to free each row first:
for (int i = 0; i < N; i++) {
    free(array_2d[i]);
}
free(array_2d);//If freed before freeing rows, memory leak occurs
```

Common Pointer Mistakes to Avoid

1. Modifying string literals

```
char *str = "COMP201";  // Read-only!
*(str+1) = 'n';  // ❌ CRASH! Modifying read-only memory
```

```
// Correct way:
char *buffer = malloc( (strlen("COMP201") + 1) * sizeof(char) ); // +1 for null terminator
strcpy(buffer, "COMP201"); // Copy string including null terminator
buffer[1] = 'n'; // Now safe to modify
```

2. Losing pointer before free

```
char *str = malloc(20);
str = "Hello";  // ❌ Lost malloc'd pointer! Memory leak!
```

3. Wrong sizeof in malloc

```
int **arr = malloc(sizeof(int) * N);  // ❌ Should be sizeof(int*)
```

Memory Leaks

What is a memory leak?

- When a program dynamically allocates memory and **forgets to later free it**, it **creates a leak**. A memory leak generally won't cause a program to misbehave, crash, or give wrong answers. A memory leak is not an urgent situation, just a little detail to eventually get around to resolving.

What is Valgrind?

- An open source system memory debugger
- Used for memory leak detection and profiling

How to use?

```
$ gcc -g -o out sample.c
```

-g	Enabling the
out	Valgrind Output file
sample.c	The program for compile

- Using -O0 is also a good idea!

- Valgrind usage:

  ```
  $ valgrind ./out
  ```

  `$ man valgrind` to play around with options

Errors that Valgrind can detect and report:

- **Invalid read/write errors**
  - Reads or writes to a memory address which you did not allocate
- **Use of an uninitialized value**
  - Code uses a declared variable before any kind of explicit assignment
- **Invalid free error**
  - Code attempts to delete allocated memory twice
  - Delete memory that was not allocated

Invalid read & writes

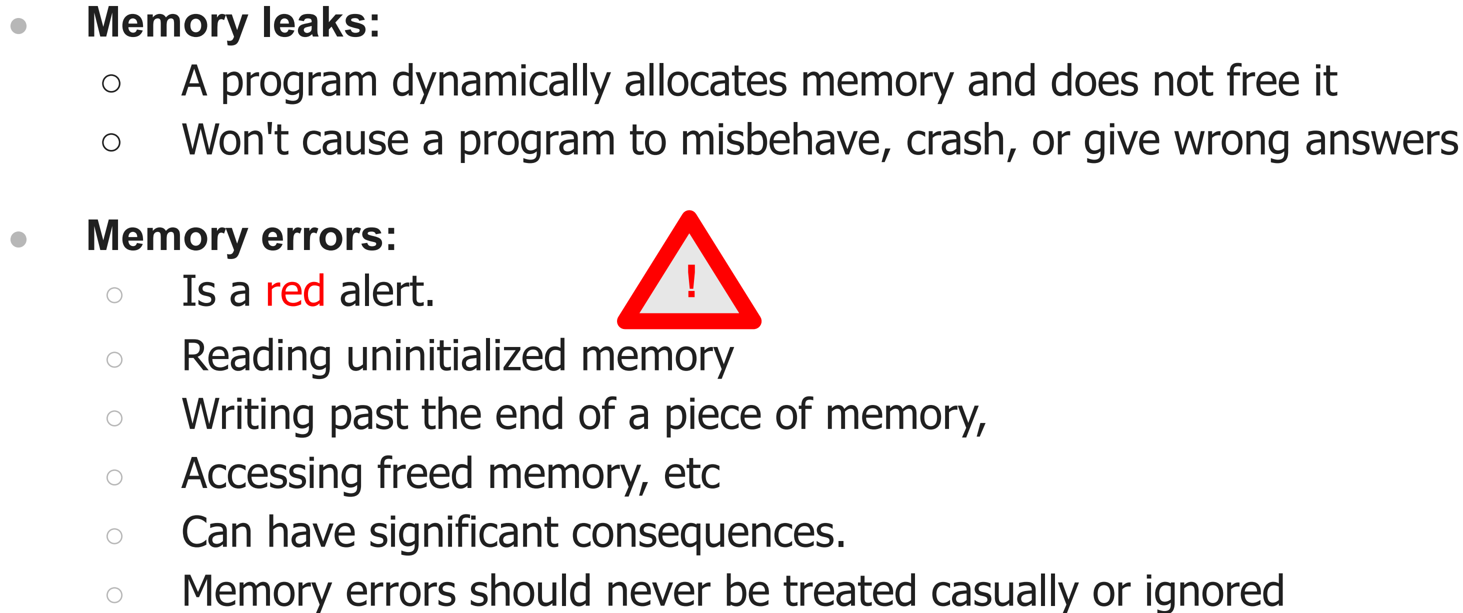
- Reading freed variables
- Reading uninitialized variables
- Writing to uninitialized memory
  - By writing too much data to allocated memory

```
int foo (int y) {

int *bar =malloc(sizeof(int));
*bar = y;

free(bar)

printf(“bar: %d \n”, * bar);
return y;

}
```

Invalid read & writes

```
==13757== Memcheck, a memory error detector
==13757== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13757== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==13757== Command: ./a.out
==13757==
bar: 32
==13757== Invalid read of size 4
==13757==    at 0x40060A: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==13757==  Address 0x5205040 is 0 bytes inside a block of size 4 free'd
==13757==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==13757==    by 0x400605: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==13757==  Block was alloc'd at
==13757==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==13757==    by 0x4005D5: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==13757==
bar: 32
==13757==
==13757== HEAP SUMMARY:
==13757==     in use at exit: 0 bytes in 0 blocks
==13757==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==13757==
==13757== All heap blocks were freed -- no leaks are possible
==13757==
==13757== For lists of detected and suppressed errors, rerun with: -s
==13757== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Memory Errors Vs. Memory Leaks



Types of Memory Leaks

- Still Reachable
  - Memory block is still pointed at, programmer could go back and free it before exiting
- Indirectly Lost
  - Block is lost because the blocks that point to it are themselves lost
- Definitely Lost
  - No pointer to the block can be found
- Possibly Lost
  - Pointer exists but it points to an internal part of the memory block

Memory Leaks

- Memory that is allocated should always be freed

```
int foo (int y) {

int *bar =malloc(sizeof(int));
*bar = y;

printf(“bar: %d \n”, * bar);
return y;

}
```

Example: sample.c

- With a memory error and a memory leak

```
$ gcc -g -o out sample.c
$ valgrind ./out
```

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

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
//
```

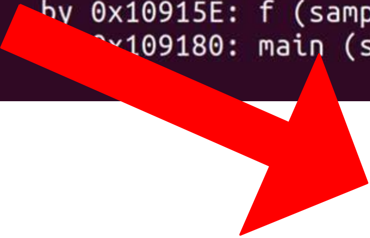
Memory error

➢ **valgrind --tool=memcheck ./out**

```
ntofighi21@njt:~/darsi/comp201/lab3$ valgrind --tool=memcheck ./out
==33826== Memcheck, a memory error detector
==33826== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==33826== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==33826== Command: ./out
==33826==
==33826== Invalid write of size 4
==33826==    at 0x10916B: f (sample.c:6)
==33826==    by 0x109180: main (sample.c:11)
==33826==  Address 0x4a57068 is 0 bytes after a block of size 40 alloc'd
==33826==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==33826==    by 0x10915E: f (sample.c:5)
==33826==    by 0x109180: main (sample.c:11)
==33826==
```

- **valgrind --tool=memcheck ./out**

```
ntofighi21@njt:~/darsi/comp201/lab3$ valgrind --tool=memcheck ./out
==33826== Memcheck, a memory error detector
==33826== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==33826== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==33826== Command: ./out
==33826==
==33826== Invalid write of size 4
==33826==    at 0x10916B: f (sample.c:6)
==33826==    by 0x109180: main (sample.c:11)
==33826==  Address 0x4a57068 is 0 bytes after a block of size 40 alloc'd
==33826==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==33826==    by 0x10915E: f (sample.c:5)
==33826==    by 0x109180: main (sample.c:11)
==33826==
```

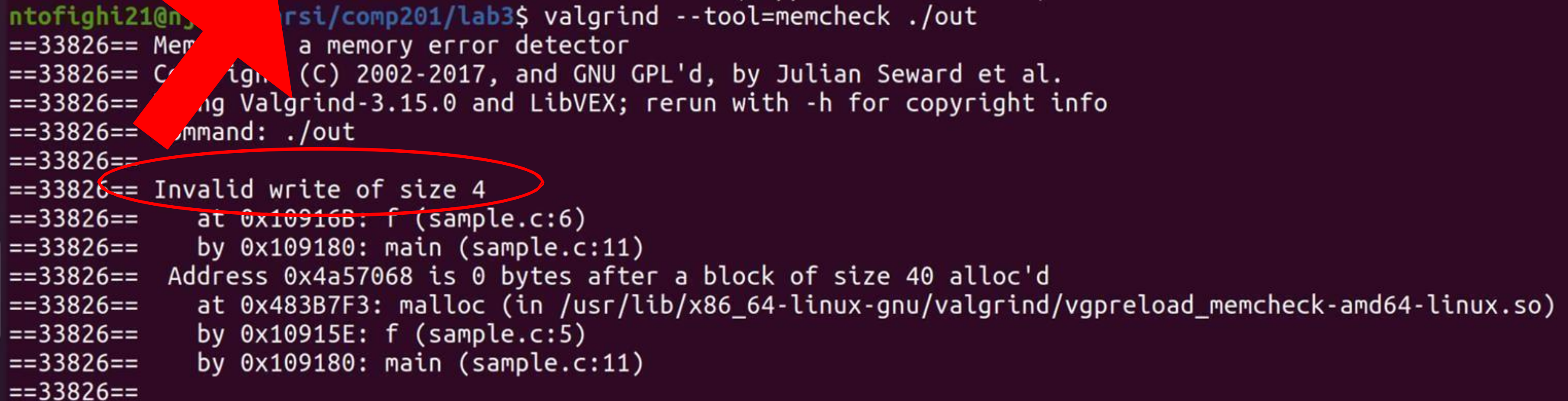


process ID

➢ **valgrind --tool=memcheck ./out**

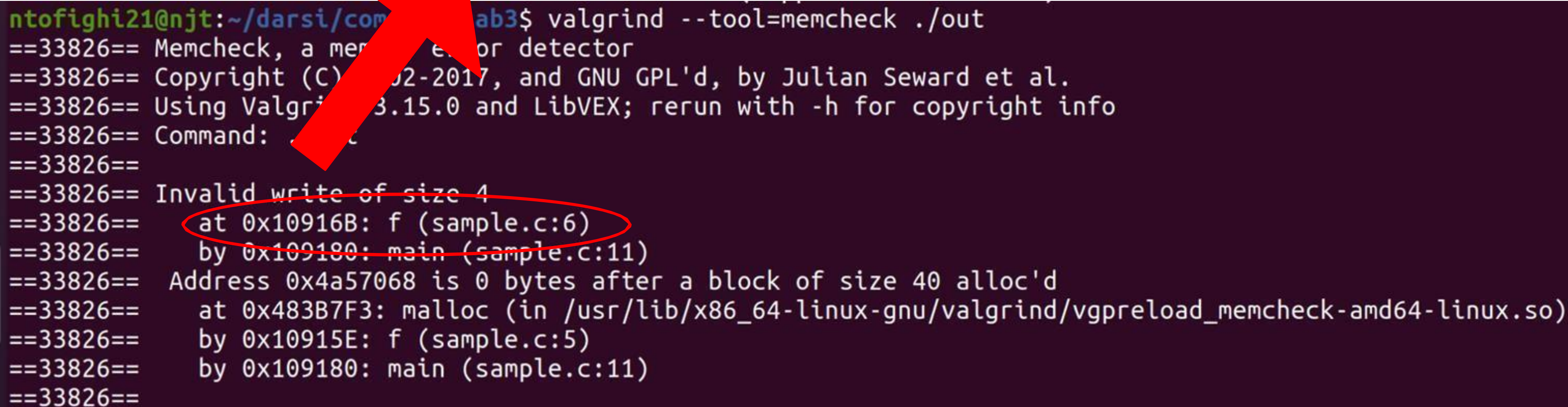
Types of error

Here; The program wrote to some memory it should not have due to a heap block overrun.



➢ **valgrind --tool=memcheck ./out**

Stack trace → where the problem occurred.



Memory error

- `valgrind --tool=memcheck --leak-check=yes ./out`

```
==40576== Memcheck, a memory error detector
==40576== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==40576== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==40576== Command: ./out
==40576==
==40576== Invalid write of size 4
==40576==    at 0x10916B: f (sample.c:6)
==40576==    by 0x109180: main (sample.c:11)
==40576==  Address 0x4a57068 is 0 bytes after a block of size 40 alloc'd
==40576==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==40576==    by 0x10915E: f (sample.c:5)
==40576==    by 0x109180: main (sample.c:11)
==40576==
==40576==
==40576== HEAP SUMMARY:
==40576==     in use at exit: 40 bytes in 1 blocks
==40576==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==40576==
==40576== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==40576==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==40576==    by 0x10915E: f (sample.c:5)
==40576==    by 0x109180: main (sample.c:11)
==40576==
==40576== LEAK SUMMARY:
==40576==    definitely lost: 40 bytes in 1 blocks
==40576==    indirectly lost: 0 bytes in 0 blocks
==40576==      possibly lost: 0 bytes in 0 blocks
==40576==    still reachable: 0 bytes in 0 blocks
==40576==         suppressed: 0 bytes in 0 blocks
==40576==
==40576== For lists of detected and suppressed errors, rerun with: -s
==40576== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Memory error

➢ **valgrind --tool=memcheck --leak-check=yes ./out**

```
=40576== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
=40576==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgprel
oad_memcheck-amd64-linux.so)
=40576==    by 0x10915E: f (sample.c:5)
=40576==    by 0x109180: main (sample.c:11)
=40576==
```

```
Memcheck, a memory error detector
opyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
 Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
d: ./out
==40576== HEAP SUMMARY:
==40576==     in use at exit: 40 bytes in 1 blocks
==40576==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==40576==
==40576== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==40576==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==40576==    by 0x10915E: f (sample.c:5)
==40576==    by 0x109180: main (sample.c:11)
==40576==
==40576== LEAK SUMMARY:
==40576==    definitely lost: 40 bytes in 1 blocks
==40576==    indirectly lost: 0 bytes in 0 blocks
==40576==      possibly lost: 0 bytes in 0 blocks
==40576==    still reachable: 0 bytes in 0 blocks
==40576==         suppressed: 0 bytes in 0 blocks
==40576==
==40576== For lists of detected and suppressed errors, rerun with: -s
==40576== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Useful links

- Valgrind and GDB in close cooperation
- Valgrind User Manual
- Valgrind memcheck tutorial
- Pointer arithmetics
- Array of pointers
- Pointers in C
- Dynamic memory alloc in C