CSE 374 Programming concepts and tools

Winter 2024

Instructor: Alex McKinney

Today

Memory Leaks

Lots of examples (+ demos)

Valgrind

Memory Errors

What is wrong here?

```
int x[] = {1, 2, 3};

free(x);
```

x is a local variable stored in stack, cannot be freed !

- The `free()` function is used to deallocate memory that was previously allocated using dynamic memory allocation functions like `malloc`, `calloc`, or `realloc`.
- The array x is declared as a regular array, not as a dynamically allocated array. It should not be freed using the free() function.

Common Memory Errors

- Dereferencing a non-pointer
- Accessing freed memory
- Double free
- Out-of-bounds access
- Reading memory before initialization
- Wrong allocation size
- Forgetting to free memory ("memory leak")

Memory Leak

A **memory leak** occurs when code fails to deallocate dynamically-allocated memory that is no longer used

- *e.g.* forget to `free` malloc-ed block, lose/change pointer to malloc-ed block

What happens: program's memory will keep growing

- This might be OK for *short-lived* program, since all memory is deallocated when program ends
- Usually has bad repercussions for *long-lived* programs
  - Might slow down over time
  - Might exhaust all available memory and crash
  - Other programs might get starved of memory

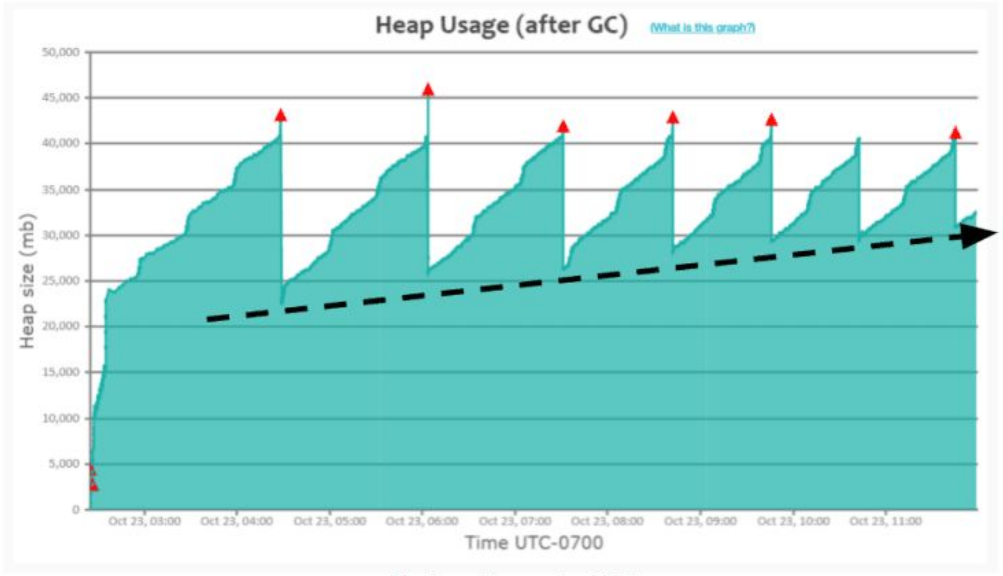
Memory Leaks in Production

Memory leaks occur in reality all of the time.

- End up looking like a sawtooth.

Developers need to resolve the issue, otherwise the system crashes.

- Restart the server periodically (bad).
- Use programming tools to discover and fix the issue (good).



Find the Bug!

Find That Bug! 1

```c
#define LEN 8
int arr[LEN];

for (int i = 0; i <= LEN; i++) {
  arr[i] = 0;
}
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type:

Fix:

No bounds checking

```c
#define LEN 8
int arr[LEN];

for (int i = 0; i <= LEN; i++) {
  arr[i] = 0;
}
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: E

Fix: i < LEN

Find That Bug! 2

```
int* foo() {
    int val = 0;

    return &val;
}
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: $\boxed{}$

Fix:

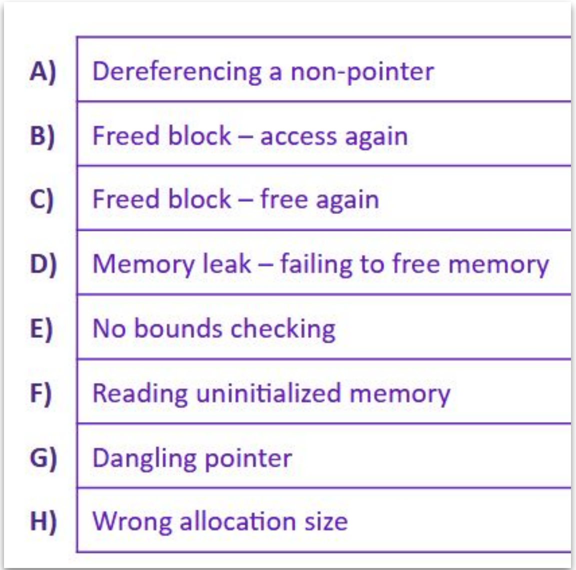
Dangling pointer

```
int* foo() {
    int val = 0;

    return &val;
}
```

Error Type: G

Fix: allocate val dynamically



A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Find That Bug! 3

```
// Create a matrix of N by M
int** p;

p = (int**)malloc(N * sizeof(int));

for (int i = 0; i < N; i++) {
    p[i] =
        (int*)malloc(M*sizeof(int));
}
```

- N and M defined elsewhere (#define)

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type:

Fix:

Wrong allocation size

```
// Create a matrix of N by M
int** p;

p = (int**)malloc(N * sizeof(int));

for (int i = 0; i < N; i++) {
   p[i] =
       (int *)malloc(M*sizeof(int));
}
```

- N and M defined elsewhere (#define)

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: H

Fix: N * sizeof(int*)

Find That Bug! 4

```
int sum_int(int* arr, int len) {
  int sum;
  for(int i = 0; i < len; i++) {
    sum += arr[i];
  }
  return sum;
}
```

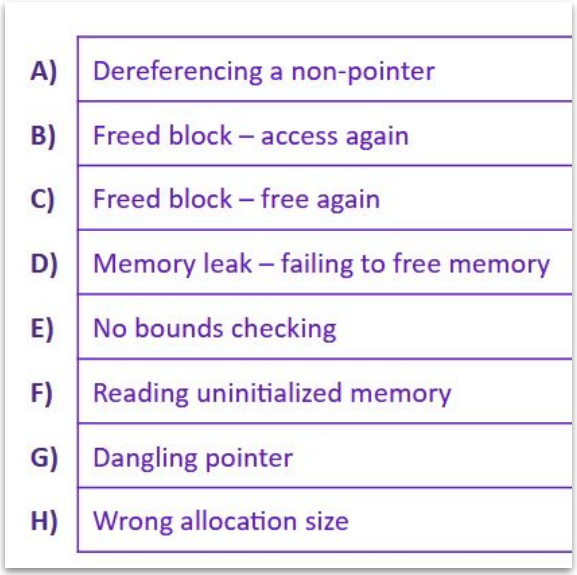
A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: ☐

Fix:

Reading uninitialized memory

```c
int sum_int(int* arr, int len) {
  int sum;
  for(int i = 0; i < len; i++) {
    sum += arr[i];
  }
  return sum;
}
```



A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: F

Fix: int sum = 0;

Aside: `scanf`

`printf` prints variables to stdout using format specifiers

`scanf` reads in values from stdin using format specifiers

- You provide `scanf` a list of addresses to store the values in

```
int age;
printf("What is your age? ");
scanf("%d", &age);
printf("You are %d years old\n", age);
```

Demo: `scanf`

Find That Bug! 5

The classic `scanf` bug

```
int scanf(const char* format, ...)
```

```
long val;

scanf("%ld", val);
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: ☐ **Fix:**

Dereferencing a non-pointer

The classic `scanf` bug

```
int scanf(const char* format, ...)
```

```
long val;

scanf("%ld", val);
```

Error Type: A

Fix: &val

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Find That Bug! 6

```
x = (int*)malloc(N*sizeof(int));
   // manipulate x
free(x);

   ...

y = (int*)malloc(M*sizeof(int));
   // manipulate y
free(x);
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: ☐

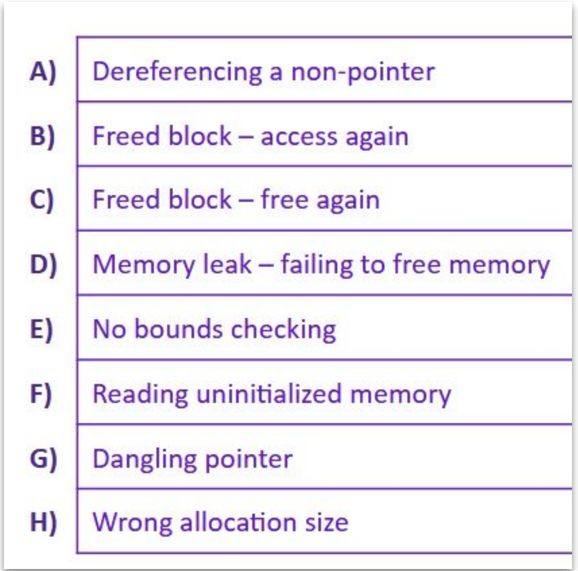
Fix:

Freed block - free again

```
x = (int*)malloc(N*sizeof(int));
   // manipulate x
free(x);

   ...

y = (int*)malloc(M*sizeof(int));
   // manipulate y
free(x);
```



A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: C

Fix: free(y);

Find That Bug! 7

```
x = (int*)malloc(M*sizeof(int));
    // manipulate x
free(x);
    // ...
y = (int*)malloc(M*sizeof(int));

for (i=0; i<M; i++) {
    y[i] = x[i];
}
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

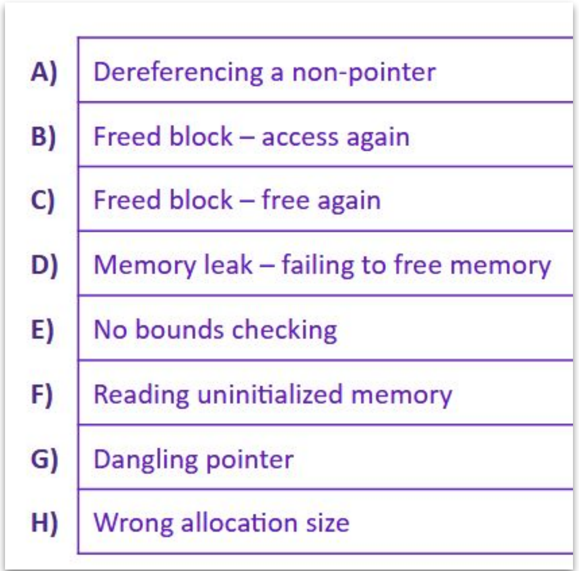
Error Type:

Fix:

Freed block – access again

```c
x = (int*)malloc(M*sizeof(int));
   // manipulate x
free(x);
   // ...
y = (int*)malloc(M*sizeof(int));

for (i=0; i<M; i++) {
   y[i] = x[i];
}
```



A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: B

Fix: free(x) after the loop

Find That Bug! 8

```
int foo() {
  int* arr = (int*)malloc(sizeof(int) * N);
  read_n_ints(N, arr);
  int sum = 0;
  for(int i = 0; i < N; i++) {
    sum += arr[i];
  }
  return sum;
}
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type:

Fix:

Memory leak - failing to free memory

```
int foo() {
  int* arr = (int*)malloc(sizeof(int) * N);
  read_n_ints(N, arr);
  int sum = 0;
  for(int i = 0; i < N; i++) {
    sum += arr[i];
  }
  return sum;
}
```

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Dangling pointer
H)	Wrong allocation size

Error Type: D

Fix: free(arr);

Questions?

Finding and Fixing Memory Errors

Valgrind is a tool that simulates your program to find memory errors

It can detect **all** of the errors we just talked about! 😲

It catches pointer errors during execution, prints summary of heap usage, including details of memory leaks

```
valgrind [options] ./myprogram args args...
```

- Useful option: `--leak-check=full`
  - Displays more detail about each memory leak

Valgrind Isn’t Perfect

Valgrind isn’t guaranteed to find *all* your memory problems.

- Depends on what the program is doing while it’s running under valgrind.
- If valgrind says no leaks are possible for a particular run, **it can only guarantee that for a particular run**.

For example, a memory leak might only manifest for a different user input!

- Always good to test with many different inputs to ensure correctness.
- More on testing later!

Demo: Valgrind

Questions?

Ex9 due Wednesday, HW4 due Sunday!

Ex9 is due before the beginning of the next lecture

- Link available on the website: https://courses.cs.washington.edu/courses/cse374/24wi/exercises/

HW4 due Sunday 11.59pm!

- Instructions on course website: https://courses.cs.washington.edu/courses/cse374/24wi/homeworks/hw4/