CSE 374 Programming Concepts and Tools

Lecture 10 - Memory Leaks

This week

Wed: Dynamic Memory Allocation

Today: Memory Leaks

Today

Reviewing `malloc` and `free`

Memory Examples and Demos

Valgrind: Detecting memory leaks

Review: `malloc()`

Usage:

```
type* var = (type*) malloc(n * sizeof(type))
```

Returns a `void*` to *uninitialized* heap memory

- Returns **`NULL`** to indicate failure
- Good practice to check result of `malloc`

Cast `void*` pointer to known type

- Put the type you want to convert to in parentheses before the variable

`sizeof()` makes code portable to different machines

Review: `free()`

Usage:

```
free(pointer);
```

Pointer must point to the *first byte* of heap-allocated memory

- Undefined behavior (often program crash) otherwise
- (Will do nothing if passed `NULL`)

Pointer is unaffected by call to free

- Defensive programming: can set pointer to `NULL` after freeing it

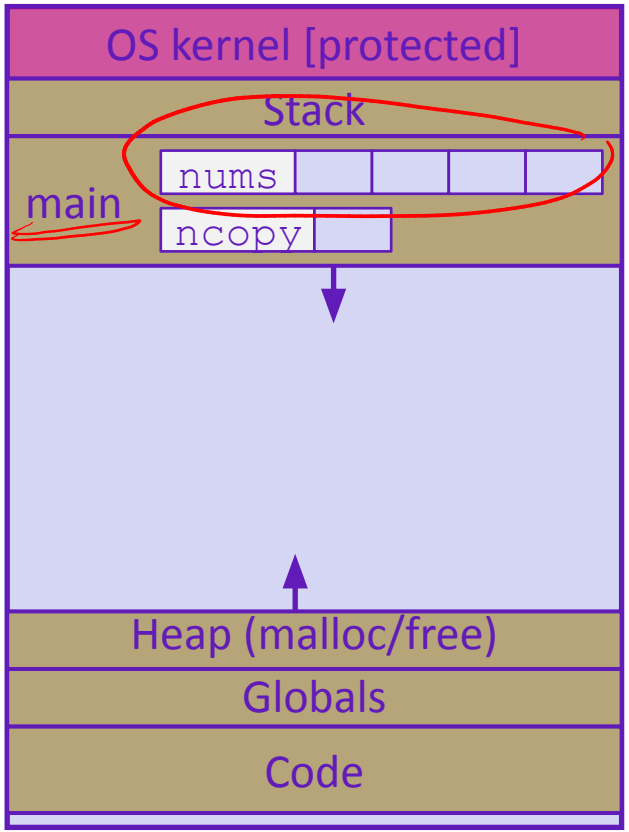
Rule of thumb: for every call to `malloc` there should be one call to `free`

Example: Heap and Stack

Initialized data

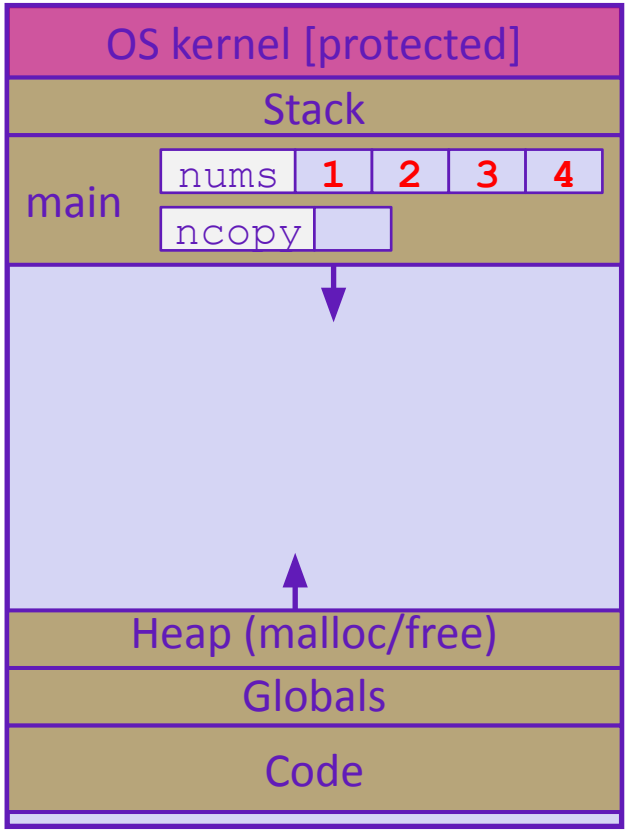
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```

Note: Arrow points to *next* line to run.



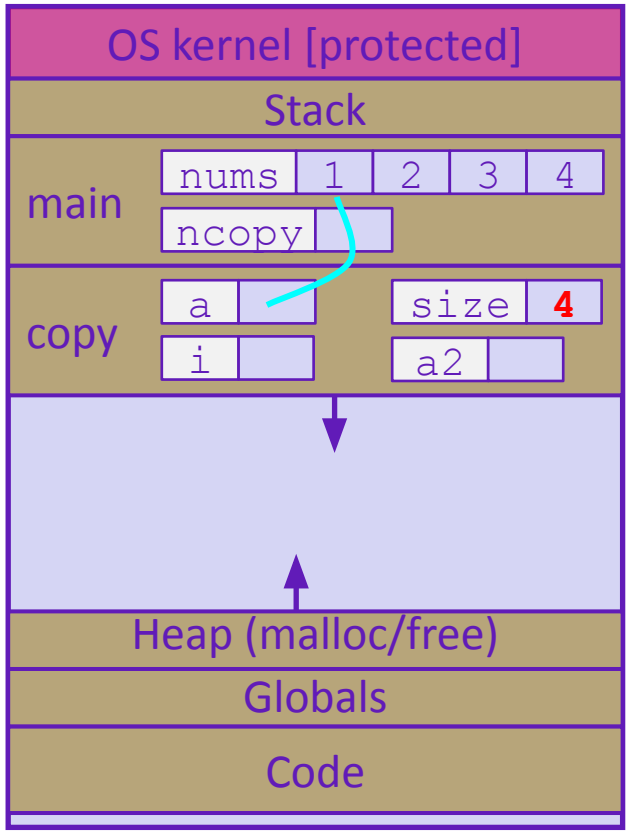
`main` local variables in stack

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



copy local variables in stack

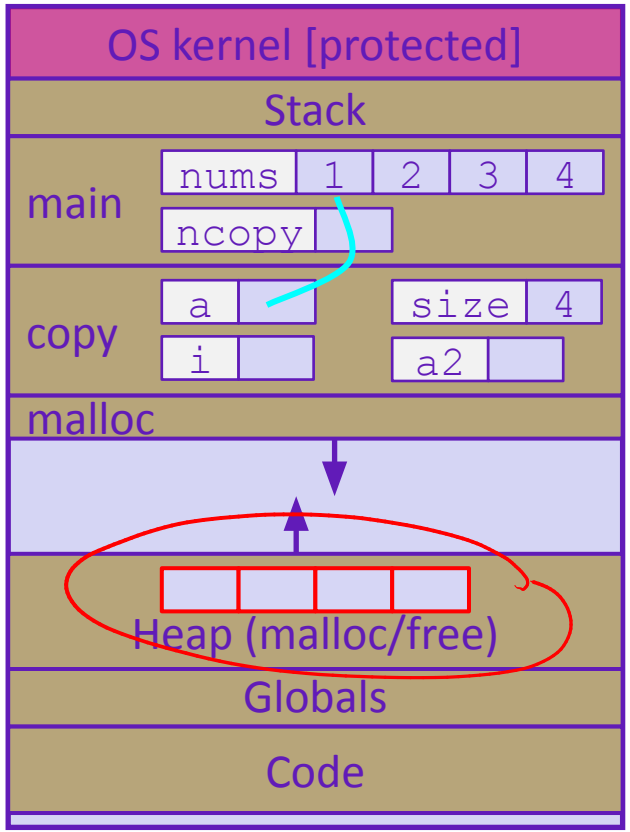
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



`malloc` space for int array

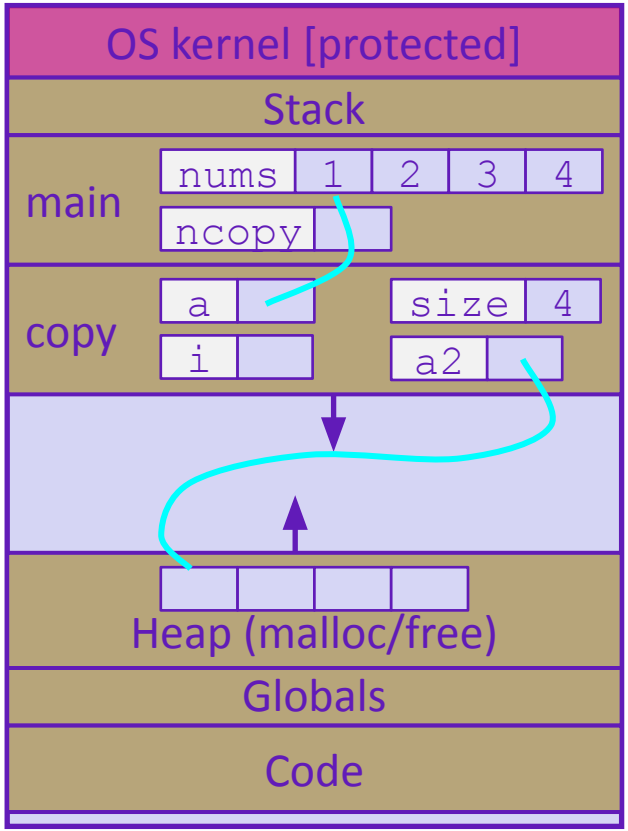
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```

4 4



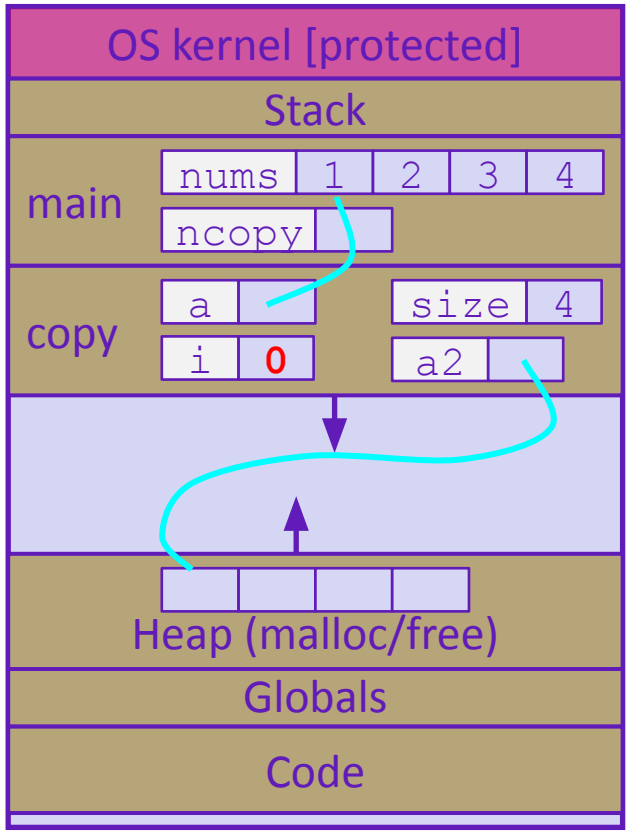
`malloc` space for int array

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



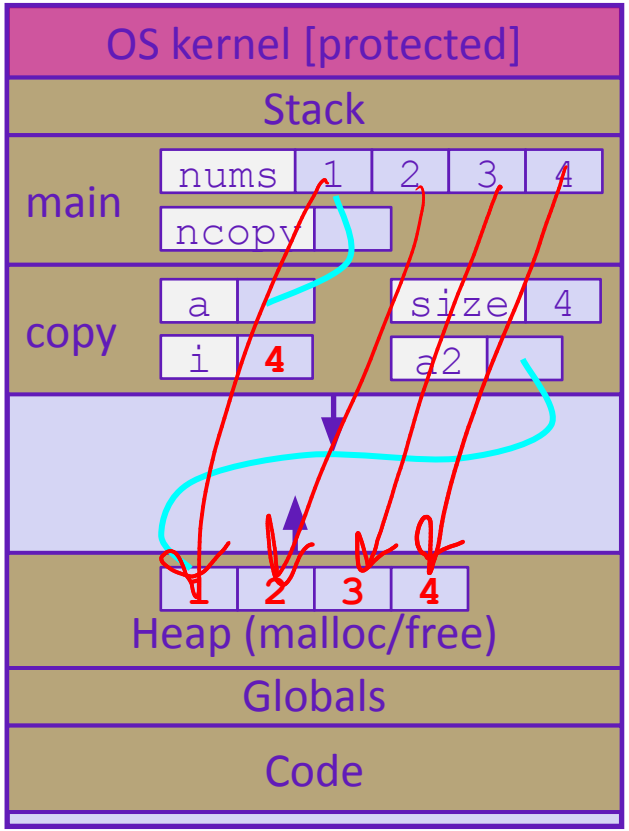
Fill available space from local variables

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



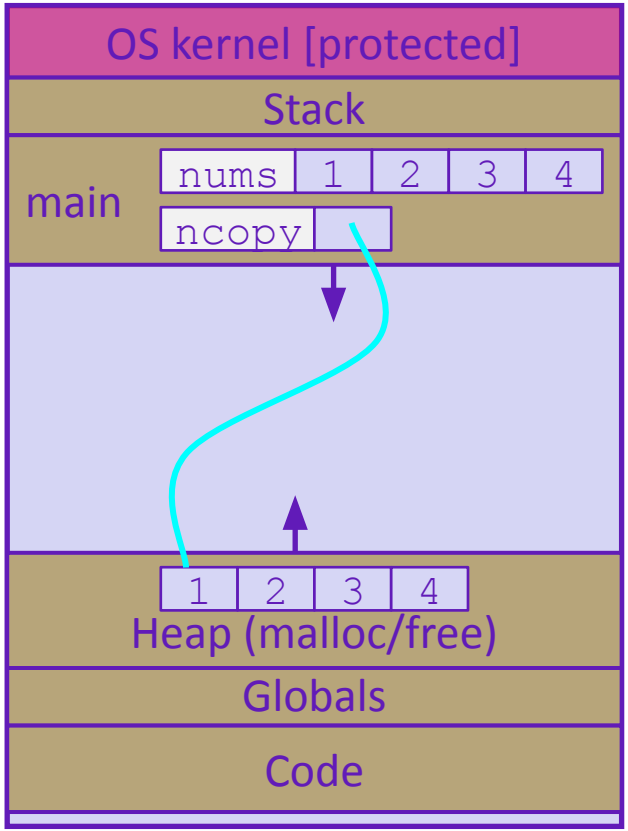
Fill available space from local variables

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



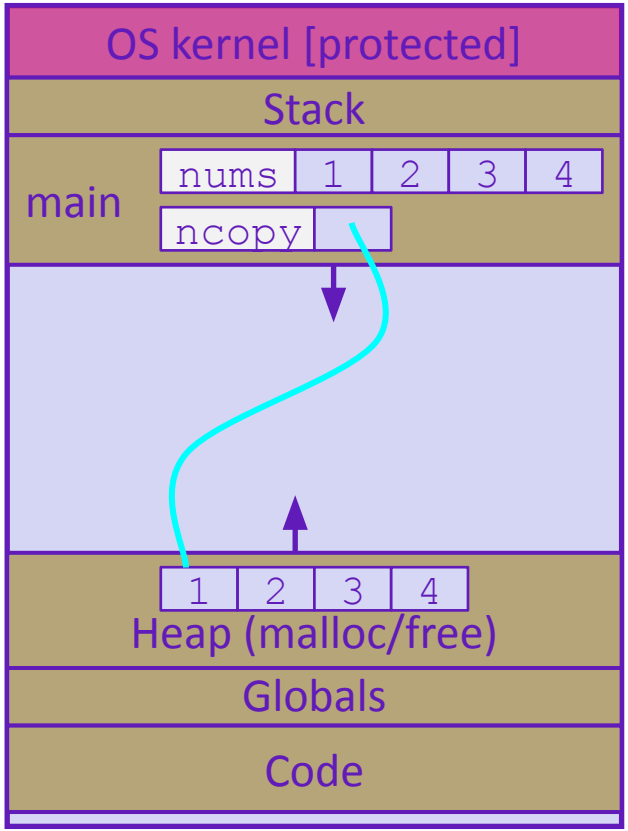
Finish copy and free stack space

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



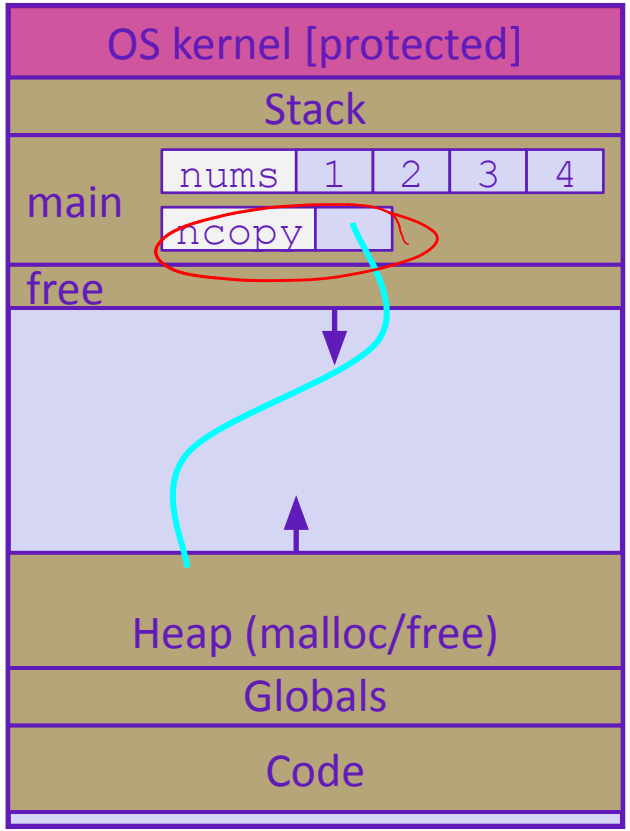
Do stuff with the array…

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



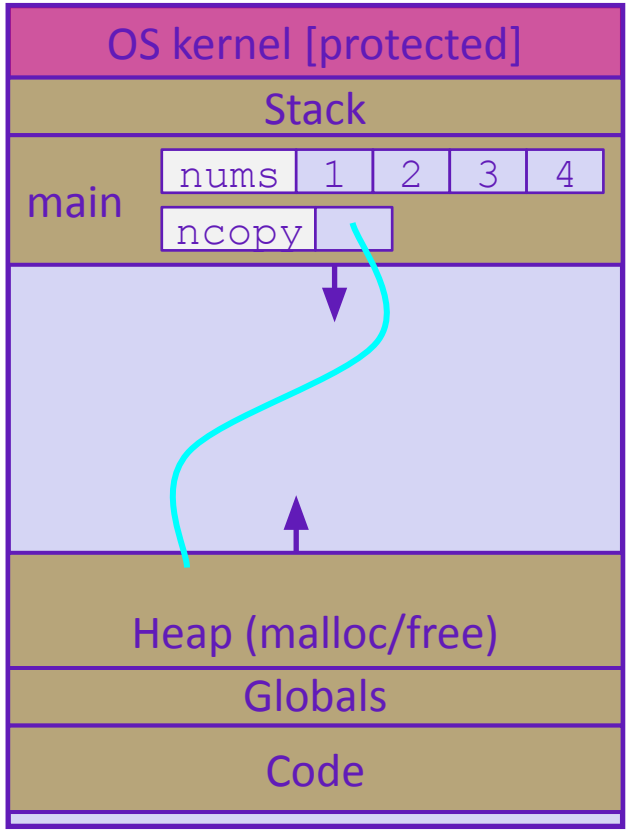
`free` ncopy from heap

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



Exit

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



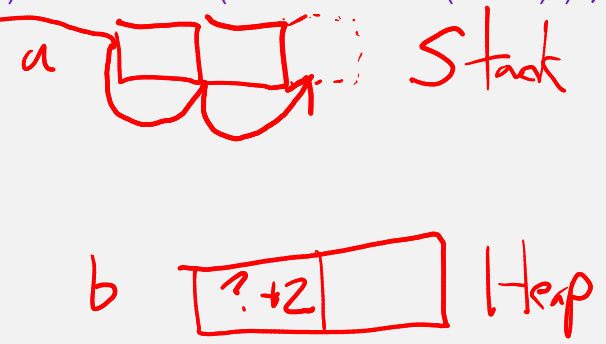
Polling Time!

Please answer in the LP10 assignment on Gradescope!

Which line(s) contain a memory error? Select *all* that apply.

- [x] Line 7 Index out of bounds
- [x] Line 8 Invalid read
- [x] Line 9
- [ ] Line 10
- [x] Line 11 Double free
- [x] Line 12 Use after free

```
1  #include <stdlib.h>
2
3  int main(int argc, char** argv) {
4    int a[2];
5    int* b = (int*) malloc(2*sizeof(int));
6
7    a[2] = 5;
8    b[0] += 2;
9    free(&(a[0]));
10   free(b);
11   free(b);
12   b[0] = 5;
13
14   return EXIT_SUCCESS;
15 }
```



Demo: `corrupt mem`

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.* forgot to `free` malloc-ed block, lost/changed pointer to malloc-ed block

The Consequences: 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

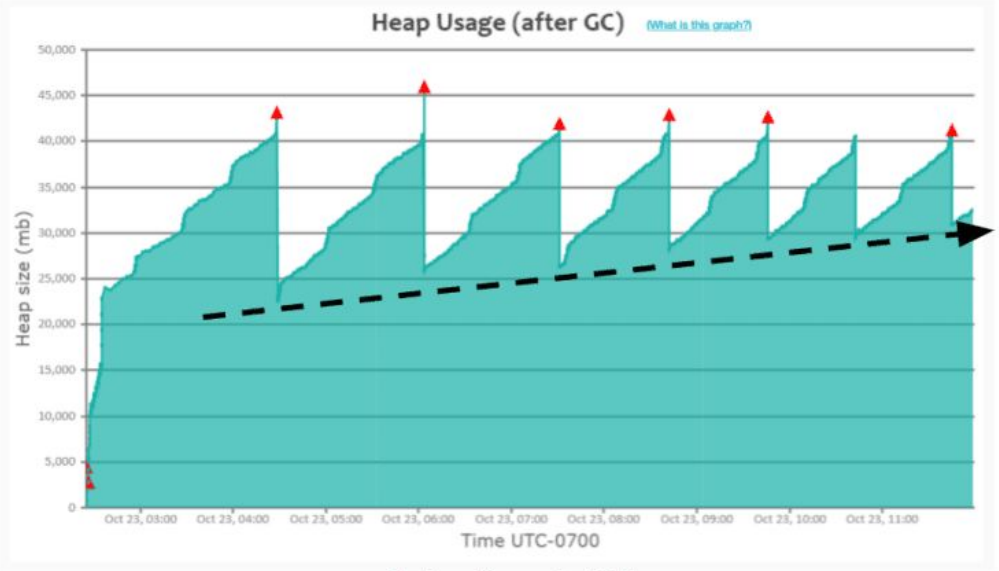
Memory Leaks in Production

Memory leaks occur in reality all of the time.

- End up looking like a sawtooth.

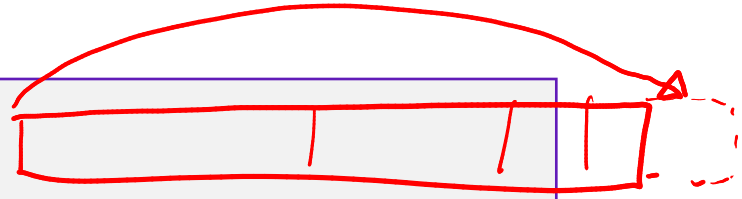
Developers need to resolve the issue, otherwise the system crashes.

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



Find the Bug!

Find That Bug! 1



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

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

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: E **Fix:**

Improper bounds checking

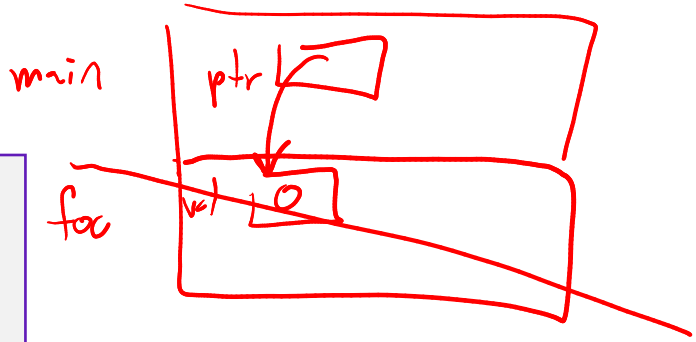
```
#define LEN 8
int arr[LEN];

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

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
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. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: G **Fix:**

Dangling pointer

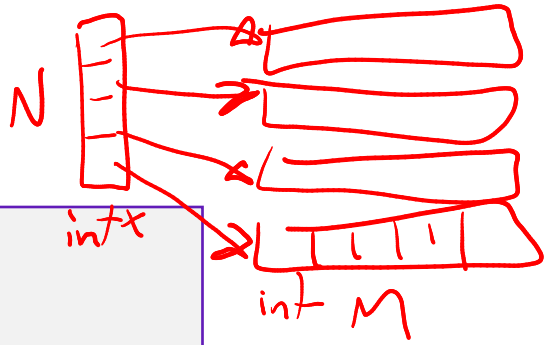
```
int* foo() {
  int val = 0;

  return &val;
}
```

```
void foo(int* val_ptr){
*val_ptr = 0;
}
```

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: G **Fix: Allocate `val` dynamically (via `malloc`)**

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));
}
```

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: H **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));
}
```

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
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;
}
```

? + = arr[i]

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: F **Fix:**

Invalid read

```
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. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
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. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
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);
```

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: A **Fix:** `&val`

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. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type:  **Fix:**

Double free

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

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

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
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 (int i = 0; i < M; i++) {
  y[i] = x[i];
}
```

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: 13 **Fix:**

Accessing already freed memory

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

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

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: **B**

Fix: move `free(x)` after `for` 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;
}
```

scanf()

free(arr);

A. Dereferencing a non-pointer
B. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: D **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. Accessing already freed memory
C. Double free
D. Memory leak – failing to free memory
E. Improper/no bounds checking
F. Invalid read
G. Dangling pointer
H. Wrong allocation size

Error Type: D **Fix: add `free(arr)` before returning**

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 specific user input!

- Always good to test with many different inputs
  - Rules out more bugs :)
- More on testing in a couple of weeks!

Demo: Valgrind

Assignment Reminders