Poll Everywhere

pollev.com/cse333sp

About how long did Exercise 1 take you?

A. **[0, 2) hours**
B. **[2, 4) hours**
C. **[4, 6) hours**
D. **[6, 8) hours**
E. **8+ Hours**
F. **I didn't submit / I prefer not to say**

Pointers, The Heap
CSE 333 Spring 2023

Instructor: Chris Thachuk

Teaching Assistants:

Byron Jin	CJ Reith
Deeksha Vatwani	Edward Zhang
Humza Lala	Lahari Nidadavolu
Noa Ferman	Saket Gollapudi
Seulchan (Paul) Han	Timmy Yang
Tim Mandzyuk	Wui Wu

Relevant Course Information (1/3)

❖ Exercise 2 out today and due Wedneday (4/5) morning

❖ Exercise grading
- Autograder scores visible immediately after deadline; sample solutions released same day as deadline
- Grades (out of 8):
  - Autograder: Compilation (1), Correctness (3), Linter (1), Valgrind (1)
  - Manual: Other Style (2)
- Style things to watch for:
  - FOLLOW THE SPEC (especially the Style Guide section)
  - Check the Google C++ Style Guide
  - Make a judgment call and document
- Keep style tips in mind, as you will need to use them in hw

Relevant Course Information (2/3)

❖ hw0 due Monday *before* 11:59 pm (and 0 seconds)

  ▪ Git: add/commit/push, then tag with `hw0-final`, then push tag

    • Then clone your repo somewhere totally different and do `git checkout hw0-final` and verify that all is well

❖ hw1 will be released by tomorrow

  ▪ You ***may not*** modify interfaces (`.h` files), but ***do*** read the interfaces while you're implementing them (!)

  ▪ Record bugs in `bugjournal.md`

  ▪ Suggestion: pace yourself and make steady progress

  ▪ Partner matching instructions will be shared on Ed

Relevant Course Information (3/3)

❖ Documentation:

- man pages, books
- Reference websites: `cplusplus.org`, `man7.org`, `gcc.gnu.org`, etc.

❖ Folklore:

- Google-ing, Stack Overflow, that rando in lab, ChatGPT

❖ Tradeoffs? Relative strengths & weaknesses?

Output Parameters

Warning: Misuse of output parameters is *the* largest cause of errors in this course!

- ❖ Output parameter
  - A pointer parameter used to store (via dereference) a function output *outside* of the function's stack frame
    - Typically points to/modifies something in the Caller's scope
  - Useful if you want to have multiple return values

- ❖ Setup and usage:
  1) Caller creates space for the data (*e.g.*, `type var;`)
  2) Caller passes in a pointer to Callee (*e.g.*, `&var`)
  3) Callee takes in output parameter (*e.g.*, `type* outparam`)
  4) Callee uses parameter to set output (*e.g.*, `*outparam = value;`)
  5) Caller accesses output via modified data (*e.g.*, `var`)

Poll Everywhere

Which is an *incorrect* way to invoke GenerateString()?

❖ Of the working ways, which would be preferred?

```
void GenerateString(char** output) {
  *output = "Hello there\n";
}
```

A.
```
char** result;
GenerateString(result);
printf("%s", *result);
```

C.
```
char* result[1] = {NULL};
GenerateString(result);
printf("%s", result[0]);
```

B.
```
char* str;
char** result = &str;
GenerateString(result);
printf("%s", str);
```

D.
```
char* result;
GenerateString(&result);
printf("%s", result);
```

E. We're lost…

Which is an *incorrect* way to invoke generateString()?

```c
void GenerateString(char** output) {
  *output = "Hello there\n";
}
```
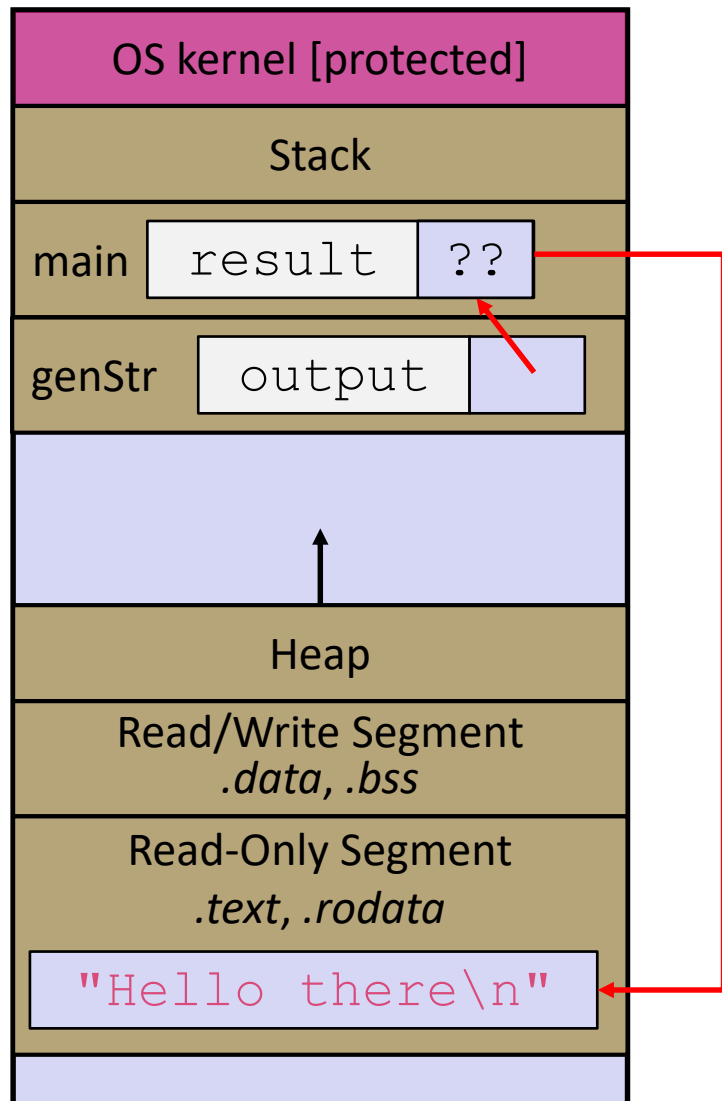
```c
char** result;
GenerateString(result);
printf("%s", *result);
```

```c
char* result[1] = {NULL};
GenerateString(result);
printf("%s", result[0]);
```

Preferred Usage

genstr.c

D.

```c
void GenerateString(char** output);

int main(int argc, char** argv) {
  char* result;
→ GenerateString(&result);
→ printf("%s", result);

  return EXIT_SUCCESS;
}


void GenerateString(char** output) {
→ *output = "Hello there\n";
→ }
```

| OS kernel [protected] |
| Stack |
| main | result | ?? |
| genStr | output | |
| |
| Heap |
| Read/Write Segment .data, .bss |
| Read-Only Segment .text, .rodata |
| "Hello there\n" |

✓ Works correctly (unlike A)

✓ Minimizes memory usage (unlike B)

✓ Intent is clear (unlike C)

Lecture Outline

- ❖ **Function Pointers**
- ❖ Heap-allocated Memory
  - ▪ `malloc()` and `free()`
  - ▪ Memory leaks

Function Pointers

❖ Based on what you know about assembly, what is a function name, really?

   ▪ Can use pointers that store addresses of functions!

❖ Generic format:

   ```
   returnType (* name)(type1, …, typeN)
   ```

   ▪ Looks like a function prototype with extra * in front of name

   ▪ Why are parentheses around `(* name)` needed?

❖ Using the function: `(*name)(arg1, …, argN)`

   ▪ Calls the pointed-to function with the given arguments and return the return value

Function Pointer Example

❖ `Map()` performs operation on each element of an array

```c
#define LEN 4

int Negate(int num) {return -num;}          // funcptr parameter
int Square(int num) {return num * num;}

// perform operation pointed to on each array element
void Map(int a[], int len, int (* op)(int n)) {
  for (int i = 0; i < len; i++) {
    a[i] = (*op)(a[i]);  // dereference function pointer
  }                      // funcptr dereference
}

int main(int argc, char** argv) {
  int arr[LEN] = {-1, 0, 1, 2};           // funcptr definition
  int (* op)(int n);  // function pointer called 'op'
  op = Square;   // function name returns addr (like array)
  Map(arr, LEN, op);                        // funcptr assignment
  ...
```

map.c

Function Pointer Example

❖ C allows you to omit & on a function name (like arrays) and omit * when calling pointed-to function

```c
#define LEN 4

int Negate(int num) {return -num;}
int Square(int num) {return num * num;}

// perform operation pointed to on each array element
void Map(int a[], int len, int (* op)(int n)) {
  for (int i = 0; i < len; i++) {
    a[i] = op(a[i]);  // dereference function pointer
  }
}

int main(int argc, char** argv) {
  int arr[LEN] = {-1, 0, 1, 2};
  Map(arr, LEN, Square);
  ...
```

implicit funcptr dereference (no * needed)

no & needed for func ptr argument

Lecture Outline

❖ Function Pointers

❖ **Heap-allocated Memory**
  ▪ **`malloc()` and `free()`**
  ▪ **Memory leaks**

Why Dynamic Allocation?

❖ Situations where static and automatic allocation aren't sufficient:

- We need memory that persists across multiple function calls but not for the whole lifetime of the program
- We need more memory than can fit on the Stack
- We need memory whose size is not known in advance
  - *e.g.,* reading file input:

```c
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

Aside: `NULL`

❖ `NULL` is a memory location that is guaranteed to be invalid

  ▪ In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*

❖ Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error

  ▪ It's better to cause a segfault than to allow the corruption of memory!

segfault.c

```c
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

`malloc()`

- ❖ General usage: `var = (type*) malloc(size in bytes)`

- ❖ **malloc** allocates an uninitialized block of heap memory of at least the requested size
  - Returns a pointer to the first byte of that memory; returns NULL if the memory allocation failed!
  - Stylistically, you'll want to (1) use `sizeof` in your argument, (2) cast the return value, and (3) error check the return value

```c
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

- ❖ Also, see **calloc()** and **realloc()**

`free()`

- ❖ Usage: `free(pointer);`

- ❖ Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.*, something previously returned by **malloc** or **calloc**)
  - Freed memory becomes eligible for future allocation
  - Freeing NULL has no effect
  - The bits stored in the pointer are *not changed* by calling free
    - Defensive programming: can set pointer to NULL after freeing it

```c
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...            // do stuff with arr
free(arr);
arr = NULL;    // OPTIONAL
```

Heap and Stack Example

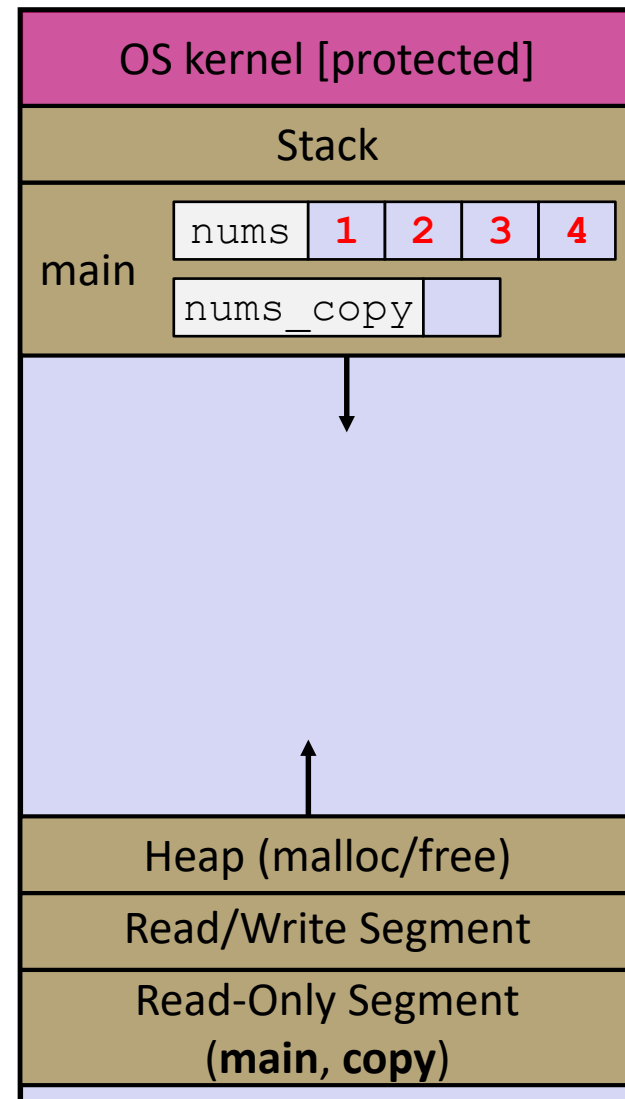arraycopy.c

```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```


OS kernel [protected]

Stack

main   nums

nums_copy

Heap (malloc/free)

Read/Write Segment

Read-Only Segment
(**main**, **copy**)

19

Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

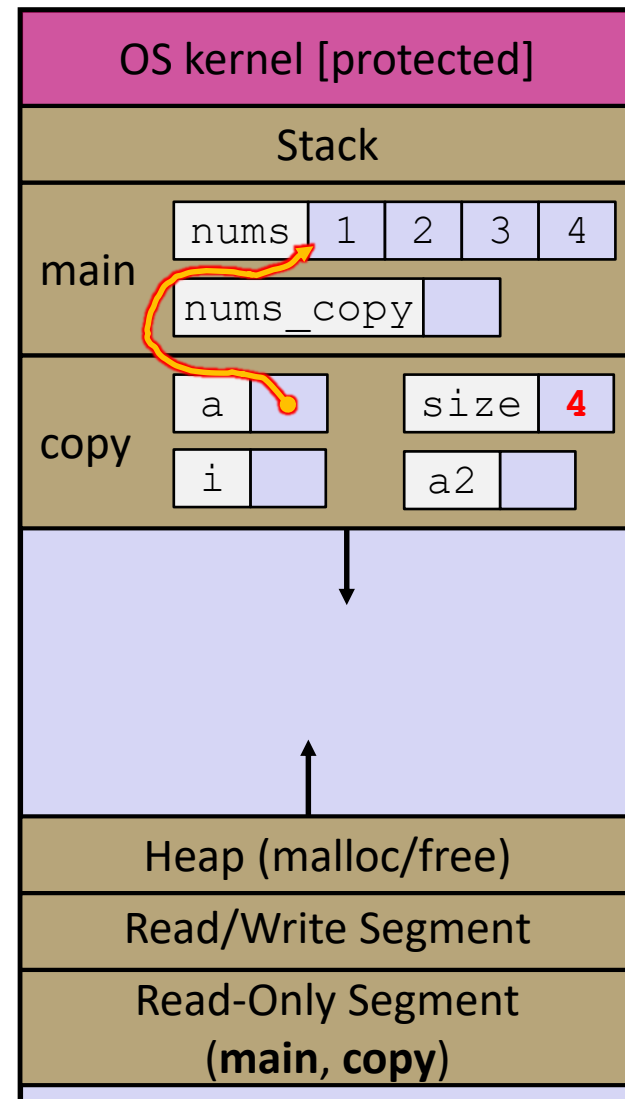```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```

| OS kernel [protected] |
| --- |
| Stack |

main

| nums | 1 | 2 | 3 | 4 |

| nums_copy | |

| Heap (malloc/free) |
| Read/Write Segment |
| Read-Only Segment (**main**, **copy**) |

# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```
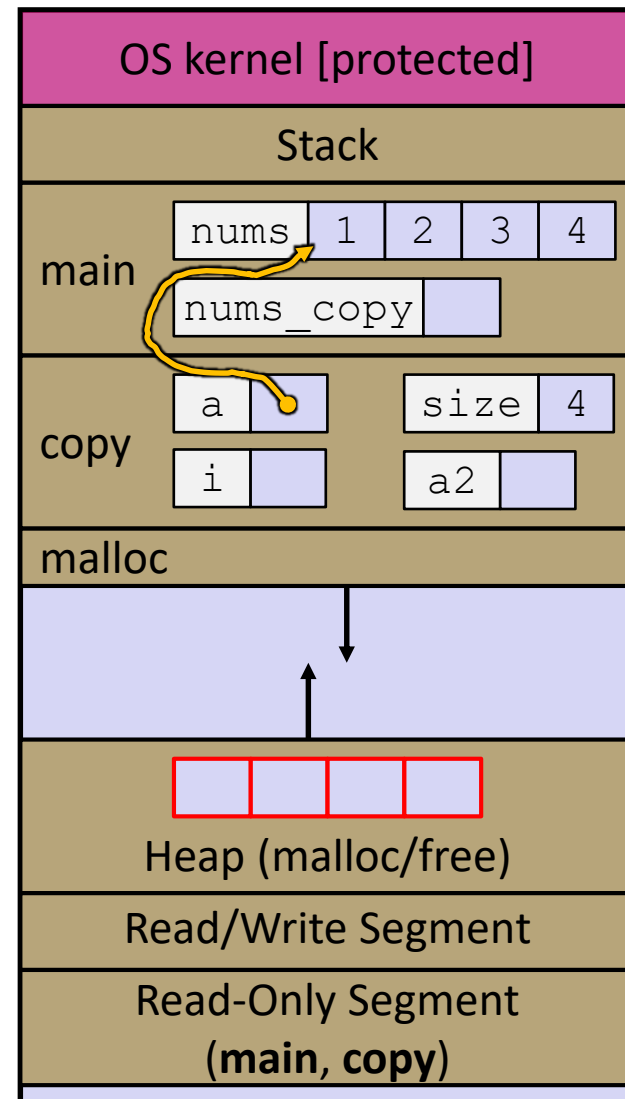

OS kernel [protected]

Stack

main

| nums | 1 | 2 | 3 | 4 |

nums_copy

copy

a   size 4

i   a2

Heap (malloc/free)

Read/Write Segment

Read-Only Segment
(**main**, **copy**)

Heap and Stack Example

Note: Arrow points to *next* instruction.

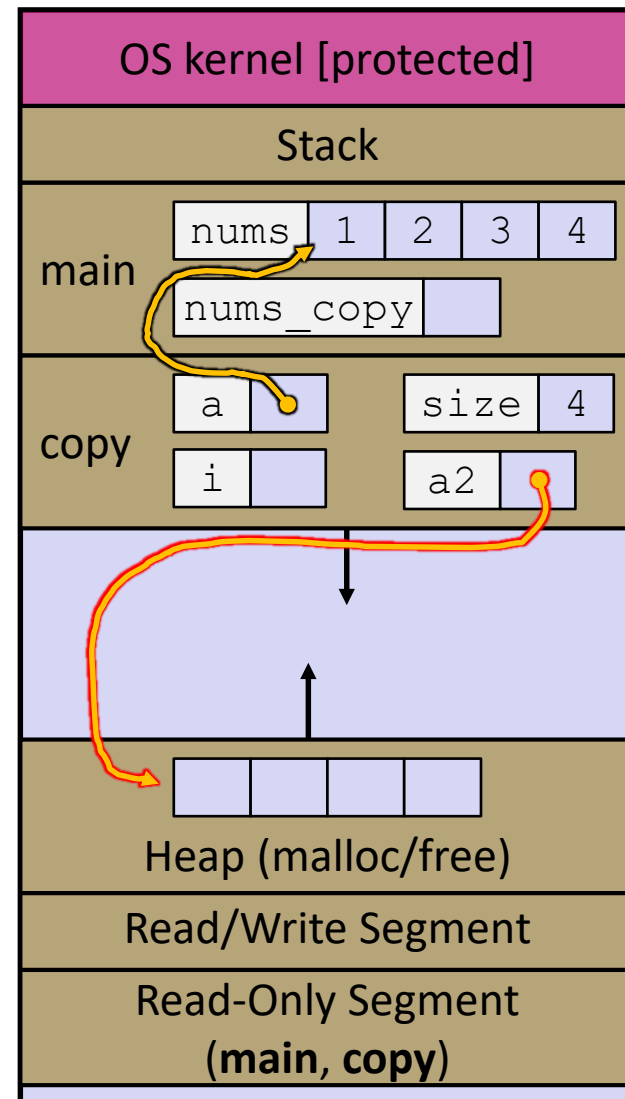arraycopy.c

```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```

Heap and Stack Example

Note: Arrow points
to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```
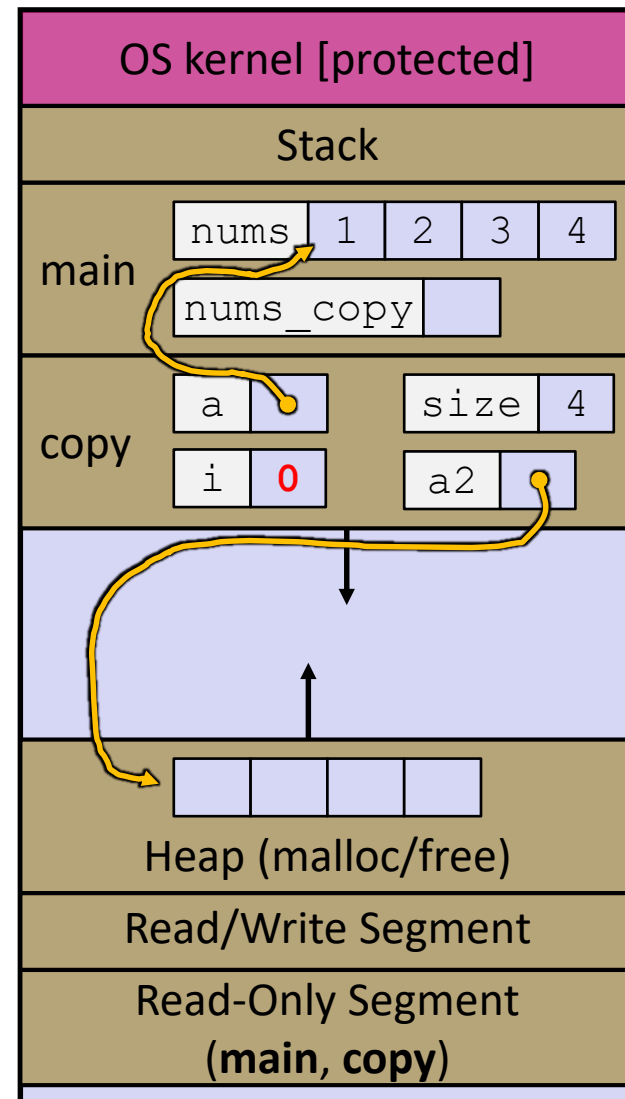
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```
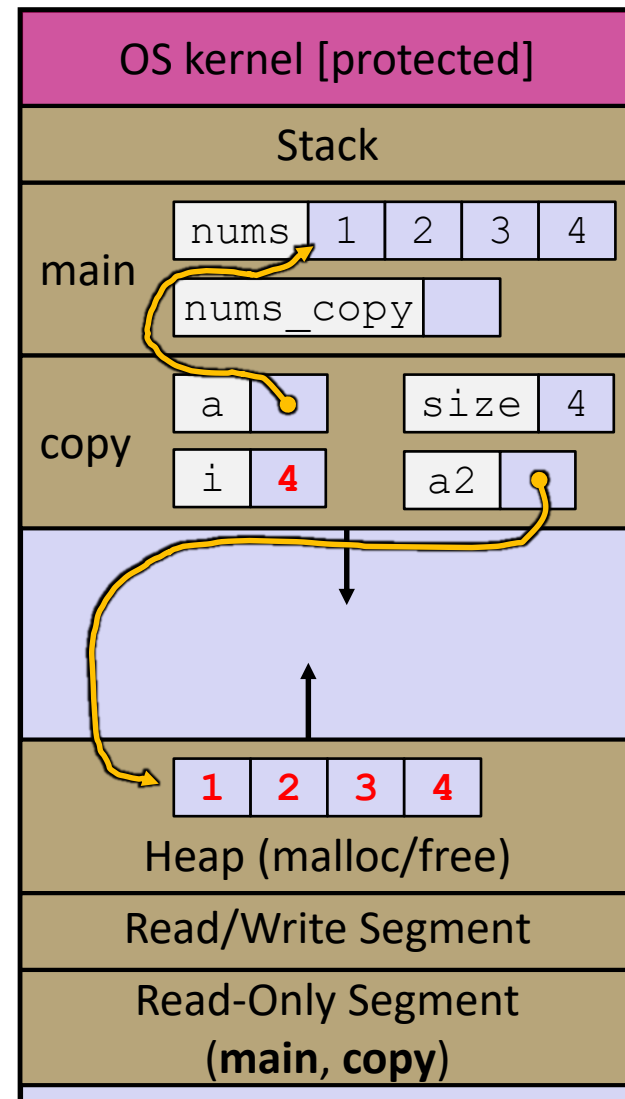

| OS kernel [protected] |
| Stack |

main
nums | 1 | 2 | 3 | 4
nums_copy

copy
a   size | 4
i | 0   a2

Heap (malloc/free)
Read/Write Segment
Read-Only Segment
(**main**, **copy**)

24

Heap and Stack Example

Note: Arrow points to *next* instruction.

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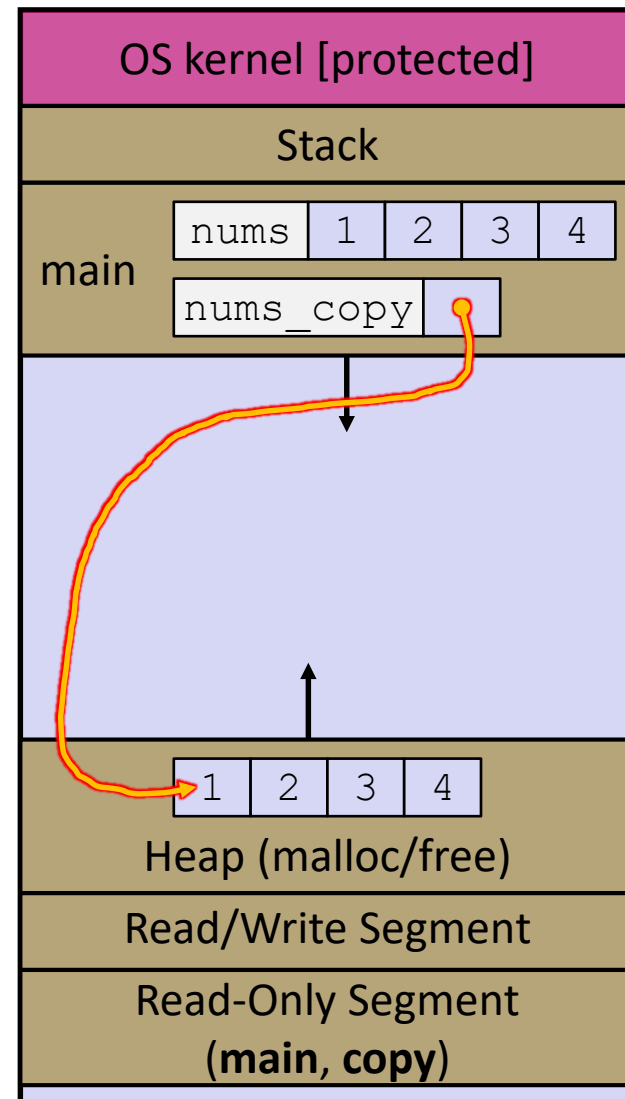
```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```


25

Heap and Stack Example

Note: Arrow points to *next* instruction.

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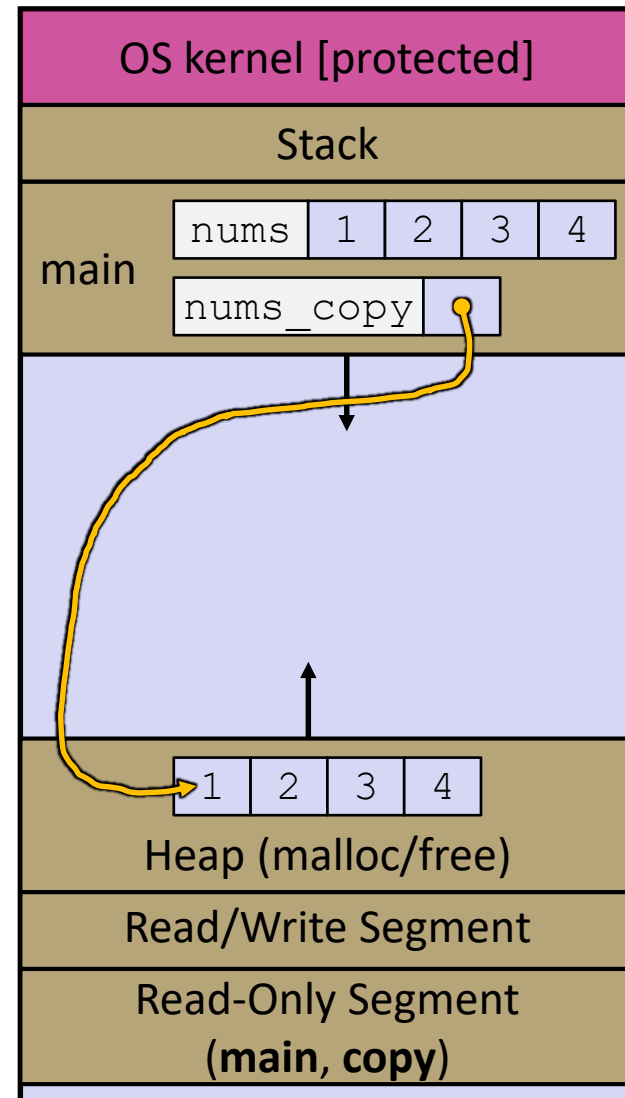
```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```

Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```


27

Heap and Stack Example

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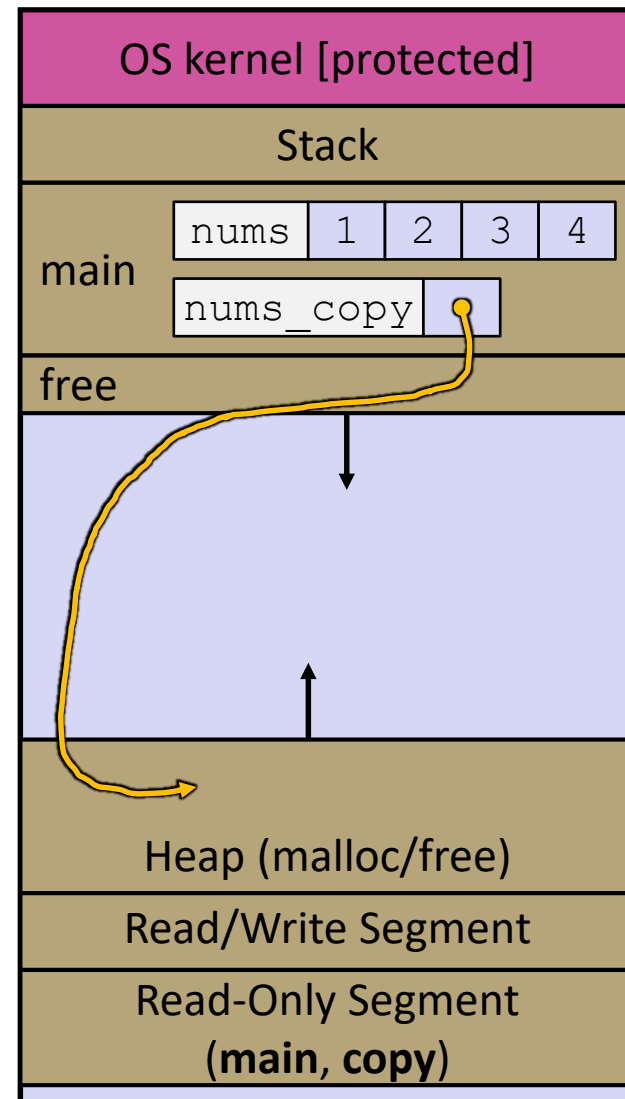
```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```

Heap and Stack Example

Note: Arrow points to *next* instruction.

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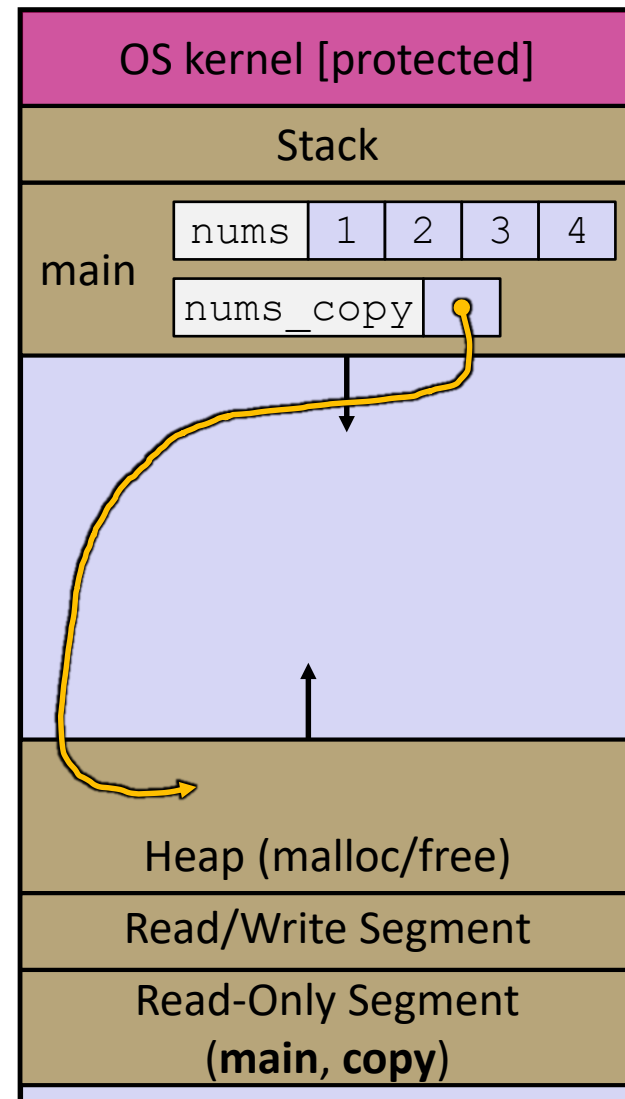
```c
#include <stdlib.h>

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

  a2 = 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```


29

Poll Everywhere

Which line will first cause a *guaranteed* error or undefined behavior?

memcorrupt.c

A. **Line 1**

B. **Line 4**

C. **Line 6**

D. **Line 7**

E. **We're lost...**

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

1  a[2] = 5;
2  b[0] += 2;
3  c = b+3;
4  free(&(a[0]));
5  free(b);
6  free(b);
7  b[0] = 5;

  return EXIT_SUCCESS;
}
```

Memory Leaks

❖ 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

■ Easier said than done; just passing pointers around – who's responsible for freeing?

❖ What happens: program's virtual memory footprint will keep growing

■ This might be OK for *short-lived* program, since all memory is deallocated when program ends

■ Usually has bad memory and performance repercussions for *long-lived* programs

Extra Exercise #1

- ❖ Write a function that:
  - Accepts a function pointer and an integer as arguments
  - Invokes the pointed-to function with the integer as its argument

Extra Exercise #2

❖ Write a function that:

- Accepts a string as a parameter

- Returns:

  - The first white-space separated word in the string as a newly-allocated string

  - AND the size of that word

Extra Exercise #3

❖ Write a function that:

- Arguments: [1] an array of ints and [2] an array length

- Malloc's an `int*` array of the same element length

- Initializes each element of the newly-allocated array to point to the corresponding element of the passed-in array

- Returns a pointer to the newly-allocated array