COMP26020 Programming Languages and Paradigms -- Part 1

Dynamic Memory Allocation

Motivation

- Many scenarios where an array size is **determined only at runtime** (vs. compile time)

```c
int main(int argc, char **argv) {

    if(argc != 2) {
        printf("Usage: %s <array size>\n",
            argv[0]);
        return -1;
    }

    int size = atoi(argv[1]);
    int array[size]; // variable size array

    for(int i=0; i<size; i++) {
        array[i] = i;
        printf("array[%d] = %d\n", i,
            array[i]);
    }

    return 0;
}
```

[11-dynamic-memory-allocation/variable-size-array.c]()

- Variable size arrays are rarely used due to support and space restrictions
- We need another solution

Program Memory Layout

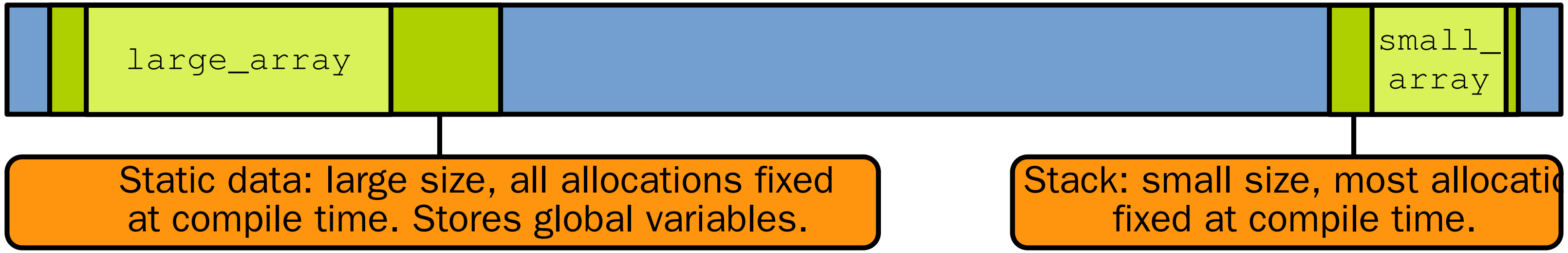
```c
int large_array[10000000];

int process_array(int *array, int size) { /* do some stuff with array */ }

int main(int argc, char **argv) {
    int size;

    /* check argc ... */

    size = atoi(argv[1]);
    if(size < 500) {            /* Make sure small_array does not overflow the stack */
        int small_array[size];
        process_array(small_array, size);
    }

    process_array(large_array, 10000000);
    return 0;
}
```

11-dynamic-memory-allocation/memory-layout.c

Program Memory Layout

```c
int large_array[10000000];

int process_array(int *array, int size) { /* do some stuff with array */ }

int main(int argc, char **argv) {
    int size;

    /* check argc ... */

    size = atoi(argv[1]);
    if(size < 500) {              /* Make sure small_array does not overflow the stack */
        int small_array[size];
        process_array(small_array, size);
    }

    process_array(large_array, 10000000);
    return 0;
}
```

11-dynamic-memory-allocation/memory-layout.c



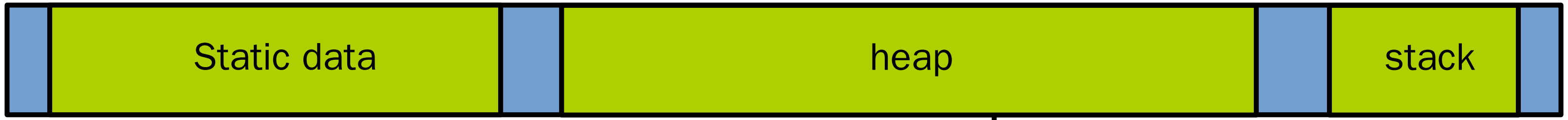
Static data: large size, all allocations fixed at compile time. Stores global variables.

Stack: small size, most allocati fixed at compile time.

Program Memory Layout

```c
int large_array[10000000];

int process_array(int *array, int size) { /* do some stuff with array */ }

int main(int argc, char **argv) {
    int size;

    /* check argc ... */

    size = atoi(argv[1]);
    if(size < 500) {            /* Make sure small_array does not overflow the stack */
        int small_array[size];
        process_array(small_array, size);
    }

    process_array(large_array, 10000000);
    return 0;
}
```

11-dynamic-memory-allocation/memory-layout.c

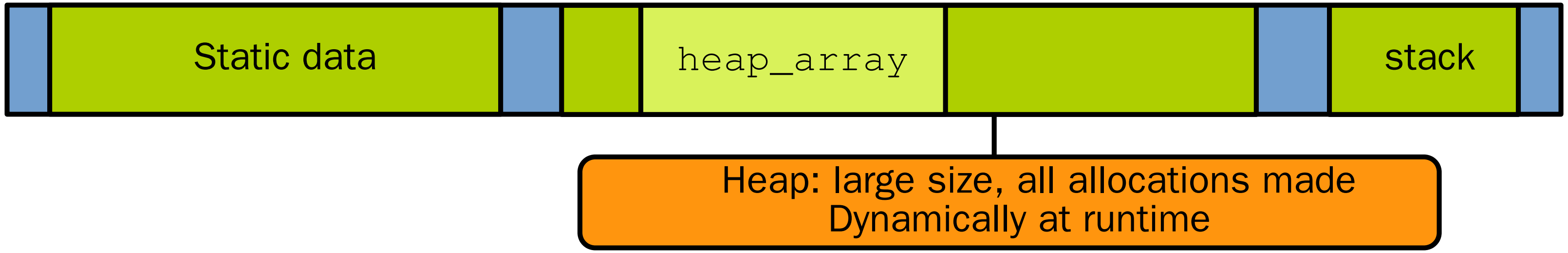


Heap: large size, all allocations made
Dynamically at runtime

Allocating on the Heap: `malloc`

```c
#include <stdio.h>
#include <stdlib.h> // needed for malloc

int process_array(int *array, int size) { /* do something with array */ }

int main(int argc, char **argv) {
    int *heap_array;
    /* ... */
    int size = atoi(argv[1]);

     heap_array = malloc(size * sizeof(int));
     if(heap_array == NULL) return -1;

    process_array(heap_array, size);

     free(heap_array);

    return 0;
}
```

11-dynamic-memory-allocation/malloc.c



Heap: large size, all allocations made
Dynamically at runtime

Allocating on the Heap: `malloc`

- **`malloc` allocates a chunk of memory on the heap and returns a pointer to it**
  - Memory is contiguous and can be used as an array
- Returns `NULL` (0) if the allocation fails
  - **Always check the return value**

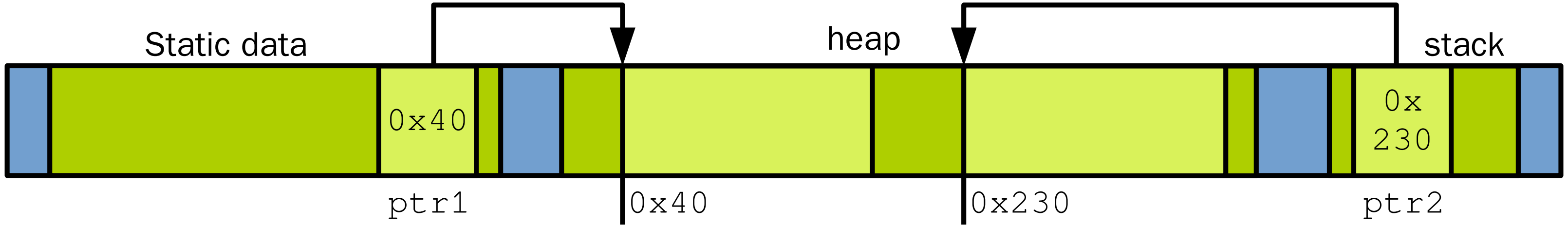
```
void *malloc(size_t size);
void free(void *ptr);
```

- `void *`: generic pointer type
- `size_t`: long unsigned int

Allocating on the Heap: `malloc`

```
double *ptr1;

int main(int argc, char **argv) {
    int *ptr2;

    ptr1 = malloc(10 * sizeof(double));
    ptr2 = malloc(30 * sizeof(int));

    if(ptr1 == NULL || ptr2 == NULL) { /* ... */ }

    /* ... */

    free(ptr1); free(ptr2);
    return 0;
}
```

[11-dynamic-memory-allocation/malloc2.c]

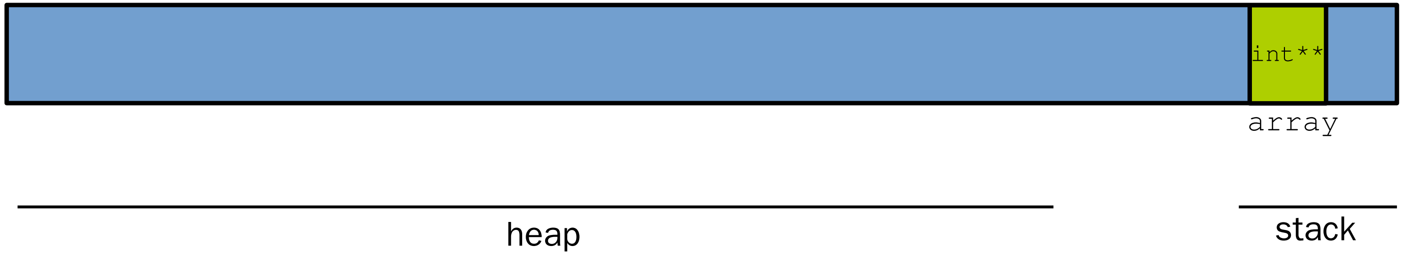


Multidimensional Arrays and `malloc`

- 3 x 2 `int` array

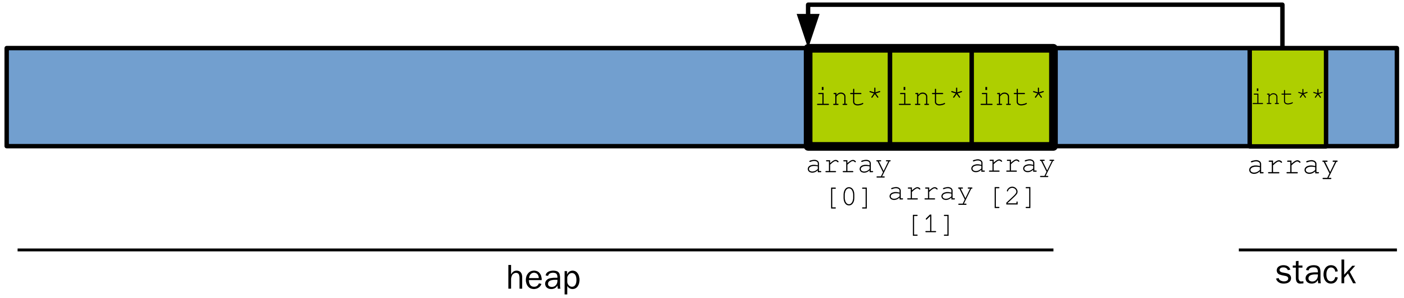
Multidimensional Arrays and `malloc`

- 3 x 2 `int` array



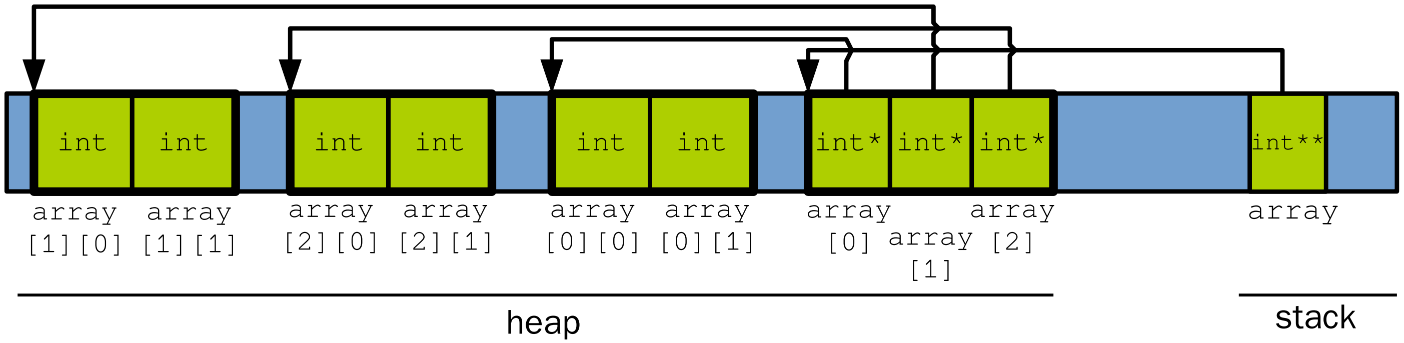
Multidimensional Arrays and `malloc`

- 3 x 2 `int` array



Multidimensional Arrays and `malloc`

- 3 x 2 `int` array



Multidimensional Arrays and `malloc`

```c
int a = 3; int b = 2;
int **array;

array = malloc(a * sizeof(int *));
if(array == NULL) return -1;

for(int i=0; i<a; i++) {
    array[i] = malloc(b * sizeof(int));
    if(array[i] == NULL) return -1;
}

array[2][0] = 12;
/* ... */

for(int i=0; i<a; i++)
    free(array[i]);
free(array);
```

11-dynamic-memory-allocation/malloc-2d-array.c

Memory Leaks

- Don't forget to free memory!

```c
void print_ten_integers() {
    int *array = malloc(10 * sizeof(int));
    if(!array) {
        printf("cannot allocate memory ...\n");
        return;
    }

    for(int i=0; i<10; i++) {
        array[i] = rand()%100;
        printf("%d ", array[i]);
    }

    printf("\n");
    /* array is neve freed, leaking 10*sizeof(int) of memory each iteration */
}

int main(int argc, char **argv) {
    int iterations = atoi(argv[1]);

    for(int i=0; i<iterations; i++)
        print_ten_integers();

    return 0;
}
```

11-dynamic-memory-allocation/leak.c

Memory Leaks

- Valgrind is a useful tool to detect memory leaks
- Let's try it on the program from the previous slide:

```
gcc -g src/leak.c -o leak
valgrind --leak-check=full ./leak 10
# ...
==11613==
==11613== HEAP SUMMARY:
==11613==     in use at exit: 400 bytes in 10 blocks
==11613==   total heap usage: 11 allocs, 1 frees, 1,424 bytes allocated
==11613==
==11613== 400 bytes in 10 blocks are definitely lost in loss record 1 of 1
==11613==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==11613==    by 0x109196: print_ten_integers (leak.c:5)
==11613==    by 0x109294: main (leak.c:32)
==11613==
==11613== LEAK SUMMARY:
==11613==    definitely lost: 400 bytes in 10 blocks
==11613==    indirectly lost: 0 bytes in 0 blocks
==11613==      possibly lost: 0 bytes in 0 blocks
==11613==    still reachable: 0 bytes in 0 blocks
==11613==         suppressed: 0 bytes in 0 blocks
```

Summary

- Malloc and free for dynamic memory allocation
- Don't forget to:
  - check malloc's return
  - free everything you malloc

Feedback form: https://bit.ly/3AsW3cZ