

Lab 4 Overview

CSCI 237: Computer Organization
Apr 3/4, 2024

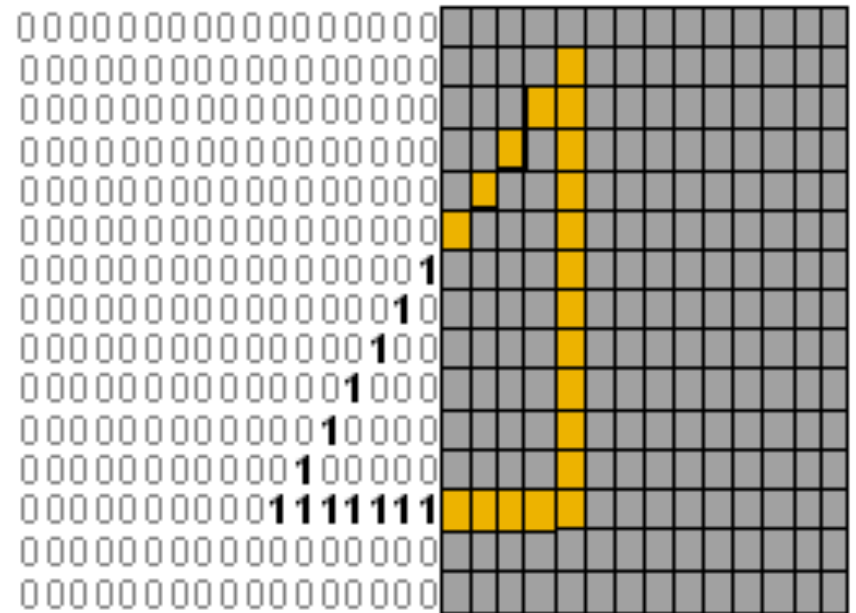
Jeannie Albrecht

Lab 4 Goals

- Finally start digging into C programming (yay!)
- Learn about malloc and free
- Learn how to use structs
- Learn about bitmaps

- Random aside:

- uint64_t values are basically just ints
- Print them with %lu



Memory Management

- Some languages automatically manage memory
 - Python and Java have *garbage collectors*
 - Run in the background to “reap” memory that is no longer being used
- We have to manage our own memory in C
 - Sounds scary, but provides us with full control of our programs
 - We allocate memory that we need, and free it when we’re finished
- Question:
 - We already know about the stack. Why isn’t this enough?

Memory Management

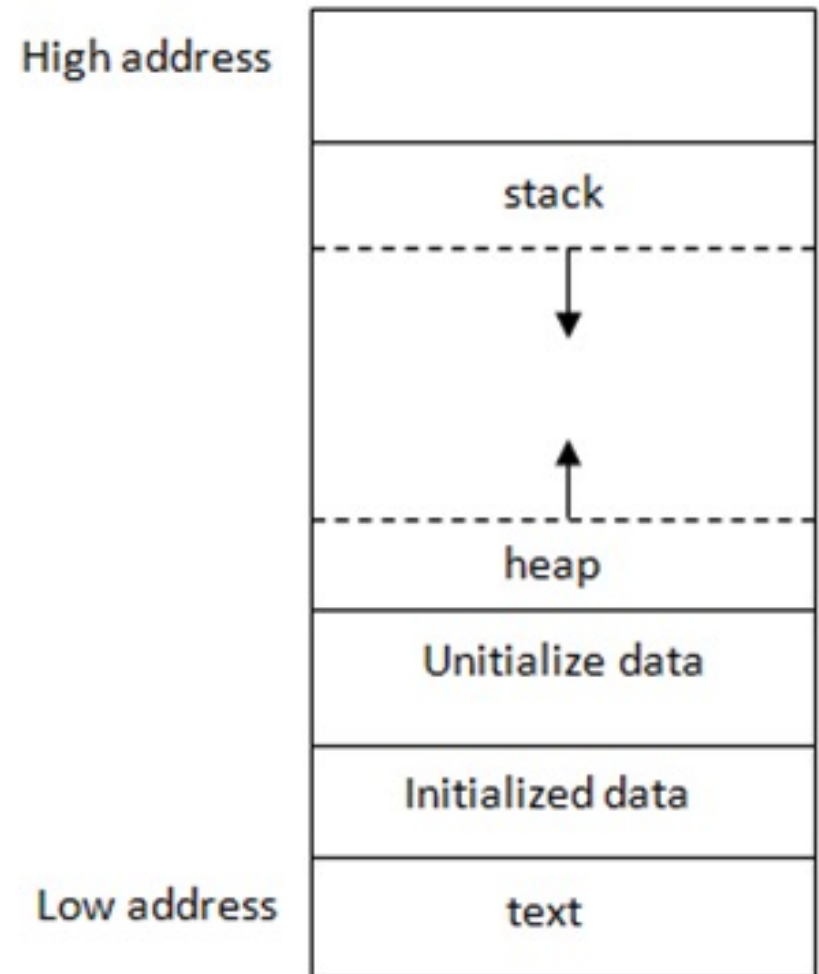
- Some languages automatically manage memory
 - Python and Java have *garbage collectors*
 - Run in the background to “reap” memory that is no longer being used
- We have to manage our own memory in C
 - Sounds scary, but provides us with full control of our programs
 - We allocate memory that we need, and free it when we’re finished
- Question:
 - We already know about the stack. Why isn’t this enough?
 - Stack frames facilitate the passing of information between function calls
 - But what happens when a function returns?

Memory Management

- Some languages automatically manage memory
 - Python and Java have *garbage collectors*
 - Run in the background to “reap” memory that is no longer being used
- We have to manage our own memory in C
 - Sounds scary, but provides us with full control of our programs
 - We allocate memory that we need, and free it when we’re finished
- Question:
 - We already know about the stack. Why isn’t this enough?
 - Stack frames facilitate the passing of information between function calls
 - But what happens when a function returns?
 - The stack is popped and memory can be used for other things!

Stack vs Heap Memory Allocation

- If we want to allocate memory for an “object” that persists beyond a stack frame, we allocate memory from a different portion of memory called the **heap**
- The heap and stack are both parts of memory
- Unlike the stack, heap memory is **dynamically** allocated and deallocated explicitly by programmers



Allocating Space in the Heap: malloc

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

int main(void) {
    char *str;
    str = malloc(100 * sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

We pass **malloc** the number of bytes we need to allocate. It returns a pointer (e.g., an address) to the beginning of that chunk of memory. In this case, the chunk has size 100 bytes.


malloc does not initialize the memory. That is done separately if needed.

Deallocating Space in the Heap: free

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

int main(void) {
    char *str;
    str = malloc(100 * sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

After we are finished with the memory, we must deallocate it, making it available for other uses. Failure to deallocate results in a memory leak. We deallocate with **free, and pass the pointer to the memory chunk being freed.**



Golden rule: Every malloc should have exactly one corresponding free.

Allocating and Deallocation structs

- structs contain one or more fields
- We have discussed struct memory allocation rules
- structs are often allocated in the heap
- Be careful when allocating structs containing pointers to other dynamically allocated structs/arrays!
- Always work from "outside in" when allocating
- Always work from "inside out" when freeing

```

struct container {
    int num;
    unsigned char *values; // a char array of size num
};

int main(int argc, char *argv[]) {
    // allocate space for container first
    struct container *contain = malloc(sizeof (struct container));
    if (contain == NULL) { //always check for malloc error
        return -1;
    }

    // get command line argument for size
    contain->num = atoi(argv[1]); // convert argument to int

    // now allocate space for char array with num values
    contain->values = malloc(contain->num * sizeof(char));
    if (contain->values == NULL) { // malloc error
        return -1;
    }

    // "clear" memory (in this case, set all values to 0)
    memset(contain->values, 0, contain->num);

    contain->values[0] = 'h';
    printf("Values: %s\n", contain->values);

    // free internal structs first, outer struct last
    free(contain->values);
    free(contain);
}

```

Notation reminder:
 Since **contain** is a pointer, we use **->** to reference the fields in the struct. If **contain** was not a pointer, you would use **.** to reference the fields.

Checking for memory leaks with valgrind

```
-> valgrind --leak-check=yes struct-malloc 5
```

```
==454843== Command: struct-malloc 5
```

```
==454843== HEAP SUMMARY:
```

```
==454843==      in use at exit: 0 bytes in 0 blocks
```

```
==454843==    total heap usage: 3 allocs, 3 frees, 1,045 bytes allocated
```

```
==454843==
```

```
==454843== All heap blocks were freed -- no leaks are possible
```

```
==454843==
```

```
==454843== For lists of detected and suppressed errors, rerun with: -s
```

```
==454843== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Checking for memory leaks with valgrind (after removing “free” to create leaks)

```
-> valgrind --leak-check=yes struct-malloc 5
```

```
==454715== Command: struct-malloc 5
```

```
==454715== HEAP SUMMARY:
```

```
==454715==      in use at exit: 21 bytes in 2 blocks
```

```
==454715== total heap usage: 3 allocs, 1 frees, 1,045 bytes allocated
```

```
==454715==
```

```
==454715== 21 (16 direct, 5 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
```

```
==454715== LEAK SUMMARY:
```

```
==454715==      definitely lost: 16 bytes in 1 blocks
```

```
==454715==      indirectly lost: 5 bytes in 1 blocks
```

```
==454715==      possibly lost: 0 bytes in 0 blocks
```

```
==454715==      still reachable: 0 bytes in 0 blocks
```

```
==454715==      suppressed: 0 bytes in 0 blocks
```

```
==454715==
```

```
==454715== For lists of detected and suppressed errors, rerun with: -s
```

```
==454715== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```