Stack

Handin 1

Frederik Le, studienummer: 202407788,
Jonas Graabæk Jakobsen, studienummer: 202405048
Simon Hesselberg, studienummer: 202405967

February 11, 2025

1 Stack

So the objective is to make a stack, that maintains first-in-last-out data structure. So, essentially it's a list we have to make. We will have a "stack_push" and a "stack_pop" functions.
The stack should have the ablillty to dynaiccly change depending on the size. So if the stack is full and we try a stack_push operation, then we need to allocate a stack double the size.
Same if the for an half empty stack, we can allocate and delete half the stack.

2 Code snippets

2.1 Function: stack_push

Description: Pushes an element into the stack. If the stack is full, it doubles its capacity before adding the new element.

Returns:

- 0 on success.
- -1 if memory allocation fails.

Listing 1: Implementation of stack_push

```c
int stack_push(struct stack *s, int x) {
    assert(s != NULL);

    // If stack is full, double its capacity
    if (s->height == s->capacity) {
        int new_capacity = s->capacity * 2;
        int *new_data = (int *)malloc((size_t)new_capacity * sizeof(int));

        if (new_data == NULL) {
            return -1; // Error: Not enough memory
        }

        // Copy existing data manually
        for (int i = 0; i < s->height; i++) {
            new_data[i] = s->data[i];
        }

        // Free old memory
        free(s->data);

        // Update stack
        s->data = new_data;
        s->capacity = new_capacity;
    }

    // Add element
    s->data[s->height] = x;
    s->height++;

    return 0; // Success
}
```

2.2 Function: stack_pop

Description: Removes the top element from the stack and stores it in dst. If the stack becomes too empty, it shrinks its capacity.

Returns:

- 0 on success.
- -1 if the stack is empty.

Listing 2: Implementation of stack_pop

```c
int stack_pop(struct stack *s, int *dst) {
    assert(s != NULL); // Validate pointer is not NULL

    if (s->height == 0) {
        printf("Stack is empty! Cannot pop.\n");
        return -1; // Error: Stack is empty
    }

    *dst = s->data[s->height - 1]; // Store value
    s->height--; // Reduce height

    // Shrink stack if it becomes too empty (under 1/4 of capacity)
    int new_capacity = s->capacity;
    if (s->height > 0 && s->height <= s->capacity / 4) {
        new_capacity = s->capacity / 2;
        if (new_capacity < 2) { // Keep minimum capacity at 2
            new_capacity = 2;
        }
    }

    // Only shrink if the capacity actually changes
    if (new_capacity < s->capacity) {
        int *new_data = (int *)malloc((size_t)new_capacity * sizeof(int));

        if (new_data == NULL) {
            return -1; // Error: Not enough memory
        }

        // Manually copy existing data
        for (int i = 0; i < s->height; i++) {
            new_data[i] = s->data[i];
        }

        // Free old memory
        free(s->data);

        // Update stack
        s->data = new_data;
        s->capacity = new_capacity;
        printf("Stack shrunk to capacity: %d\n", s->capacity);
    }

    return 0; // Success
}
```

3 Memory

Unlike languages such as Java, **C does not have automatic garbage collection**. This means that programmers must **manually allocate and free memory** to avoid memory leaks and inefficiencies. As our professor says, "C assumes you know what you are doing," which is rarely the case. In C, when memory is allocated dynamically, it must be explicitly **freed** once it is no longer needed. Failure to do so can result in **memory leaks**, where memory is allocated but never returned to the system.

There are multiple ways we can manipulate memory allocation.

- **malloc(size_t size)**: Allocates a specified number of bytes in memory but does not initialize them.
- **calloc(size_t num, size_t size)**: Allocates memory for an array of elements and initializes all bytes to zero.
- **realloc(void *ptr, size_t new_size)**: Resizes previously allocated memory to a new size.
- **free(void *ptr)**: Releases previously allocated memory back to the system.

In the hand-in, we are not allowed to use `realloc`, which is a bit annoying since it would be a nicer solution, but it is what it is. In code snippets, you can see that we use `malloc` to allocate memory, and when we need to resize, we copy and paste everything over to a larger or smaller stack, depending on operations.