Data Structure Intro - Dynamic Array Stack

Stacks

- When you add something to the stack, it’s pushed on top of stack
- When you take something off the the stack, it’s popped off

Header file

```
#define INIT_SIZE 10
#define ERROR_MEMORY 1
```

- The macros helps use avoid using magic numbers
- INIT_SIZE
  - The number of things our stack can store will start at 10
  - We'll then dynamically adjust as we need to
- ERROR_MEMORY
  - memory associated error code
  - if we fail to malloc() / realloc() we'll return with this error code

Construction of the stack

```c
struct stack {
    long long int top;
    size_t size;
    char *stack_data;
};
```

- A structure to keep track of the different components of the stack
- top
    - An array is used
    - Top is the index of the array
- size
    - size of the array
- stack_data
    - array storing characters

Creating a Stack

```c
struct stack create_stack() {
    struct stack my_stack;

    my_stack.top = -1;
    my_stack.size = INIT_SIZE;
    my_stack.stack_data = malloc(INIT_SIZE * sizeof(char));

    if (my_stack.stack_data == NULL) {
        printf("Unable to allocate memory!\n");
        exit(ERROR_MEMORY);
    }

    return my_stack;
}
```

- Create and initialize a stack for use
- Create an instance of stack structure
- The stack begins with a size of INIT_SIZE
- Start the top index at negative one
    - since that is not a valid array position
- Dynamically allocate memory for the stack
    - exit with ERROR_MEMORY value if malloc() returns a NULL pointer

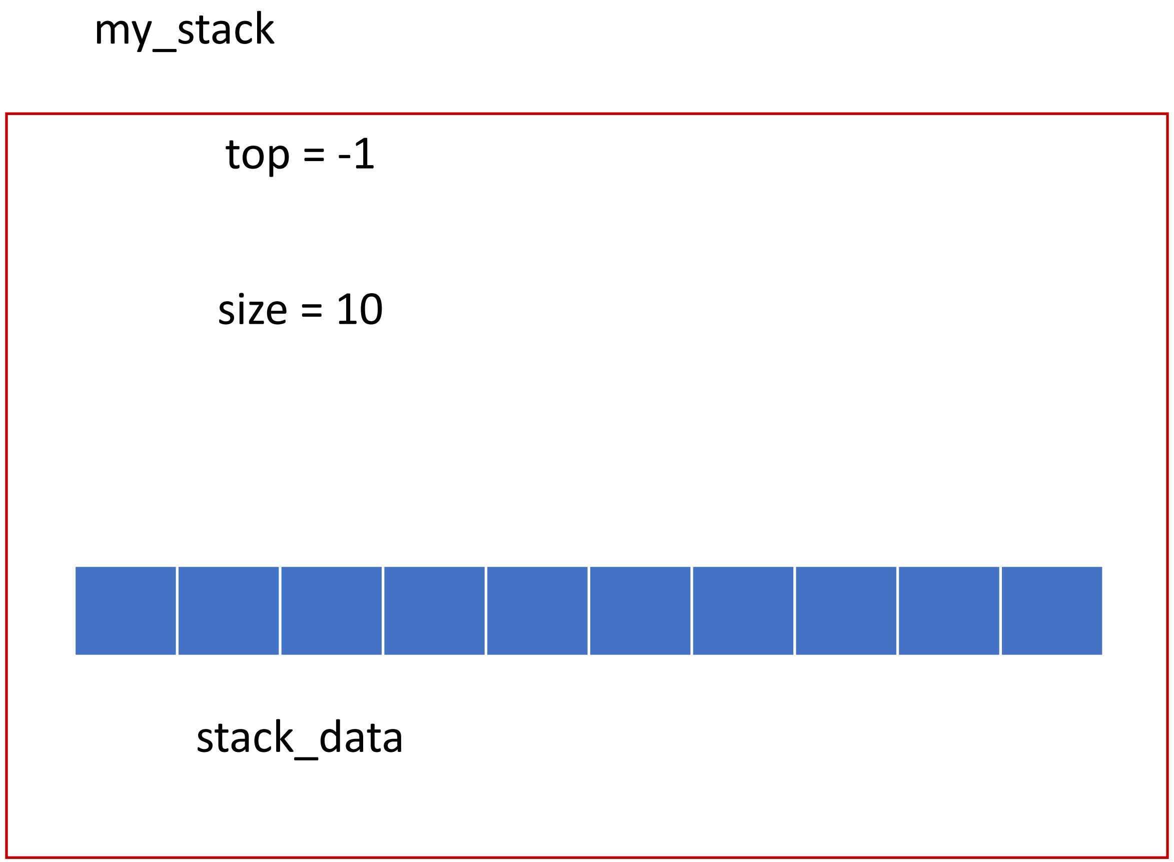
Creating a Stack

```c
struct stack create_stack() {
    struct stack my_stack;

    my_stack.top = -1;
    my_stack.size = INIT_SIZE;
    my_stack.stack_data = malloc(INIT_SIZE * sizeof(char));

    if (my_stack.stack_data == NULL) {
        printf("Unable to allocate memory!\n");
        exit(ERROR_MEMORY);
    }

    return my_stack;
}
```



Pushing items on top of stack

```
void push(struct stack *my_stack, char item) {

    if (my_stack->top == my_stack->size - 1) {
        my_stack->size *= 2;
        my_stack->stack_data = realloc(my_stack->stack_data, my_stack->size * sizeof(char));
        if (my_stack->stack_data == NULL) {
            printf("Unable to re-allocate memory!\n");
            exit(ERROR_MEMORY);
        }
    }

    ++(my_stack->top);

    my_stack->stack_data[my_stack->top] = item;
}
```

- Parameters
  - Pointer to struct stack
  - Item we want to push

Pushing items on top of stack

```
void push(struct stack *my_stack, char item) {

    if (my_stack->top == my_stack->size - 1) {
        my_stack->size *= 2;
        my_stack->stack_data = realloc(my_stack->stack_data, my_stack->size * sizeof(char));
        if (my_stack->stack_data == NULL) {
            printf("Unable to re-allocate memory!\n");
            exit(ERROR_MEMORY);
        }
    }

    ++(my_stack->top);

    my_stack->stack_data[my_stack->top] = item;
}
```

- Start by checking whether stack is full
  - top of the stack would be equal to the last index of array
- If stack is full
  - double the current size
  - dynamically resize array using realloc()

Pushing items on top of stack

```
void push(struct stack *my_stack, char item) {

    if (my_stack->top == my_stack->size - 1) {
        my_stack->size *= 2;
        my_stack->stack_data = realloc(my_stack->stack_data, my_stack->size * sizeof(char));
        if (my_stack->stack_data == NULL) {
            printf("Unable to re-allocate memory!\n");
            exit(ERROR_MEMORY);
        }
    }

    ++(my_stack->top);

    my_stack->stack_data[my_stack->top] = item;
}
```

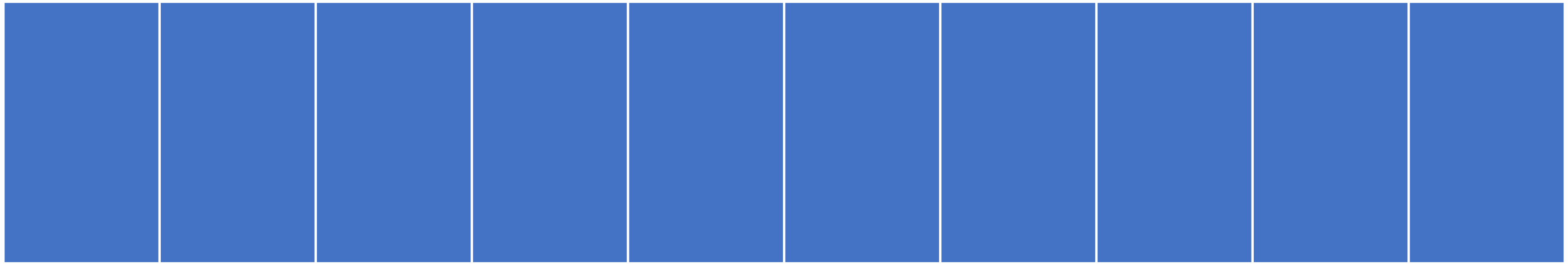
- Increment stack top
- Assign element to stack top

Pushing items on top of stack

my_stack

top = -1

size = 10

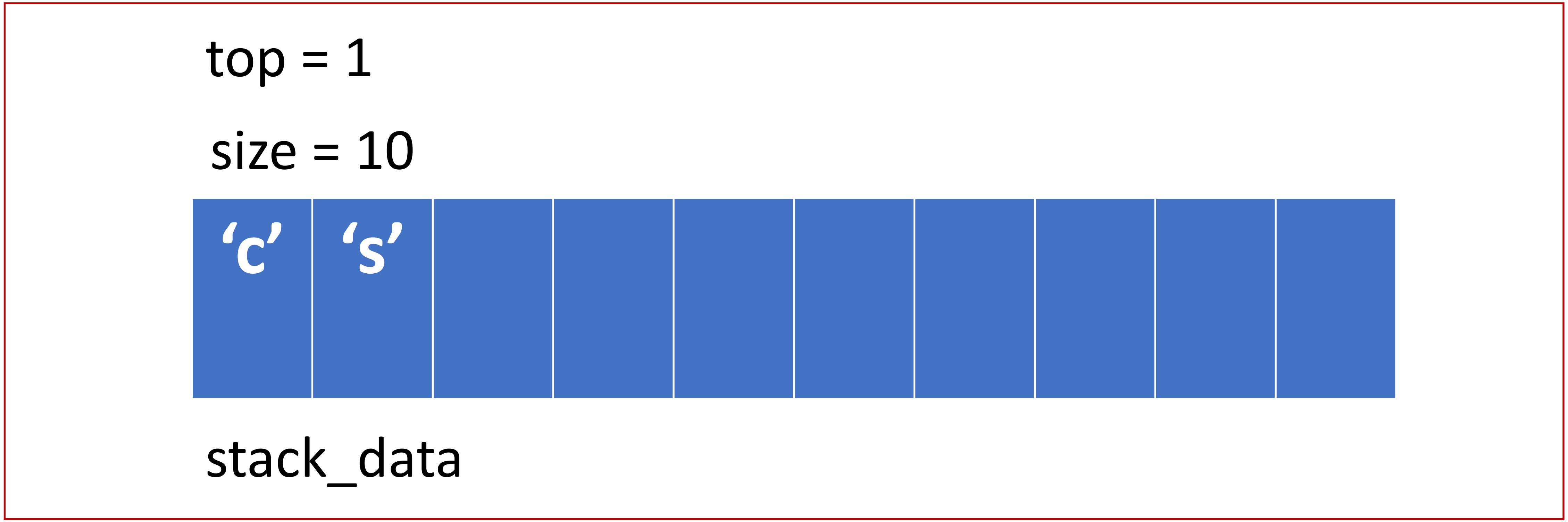


stack_data

push(&my_stack, ’c’)

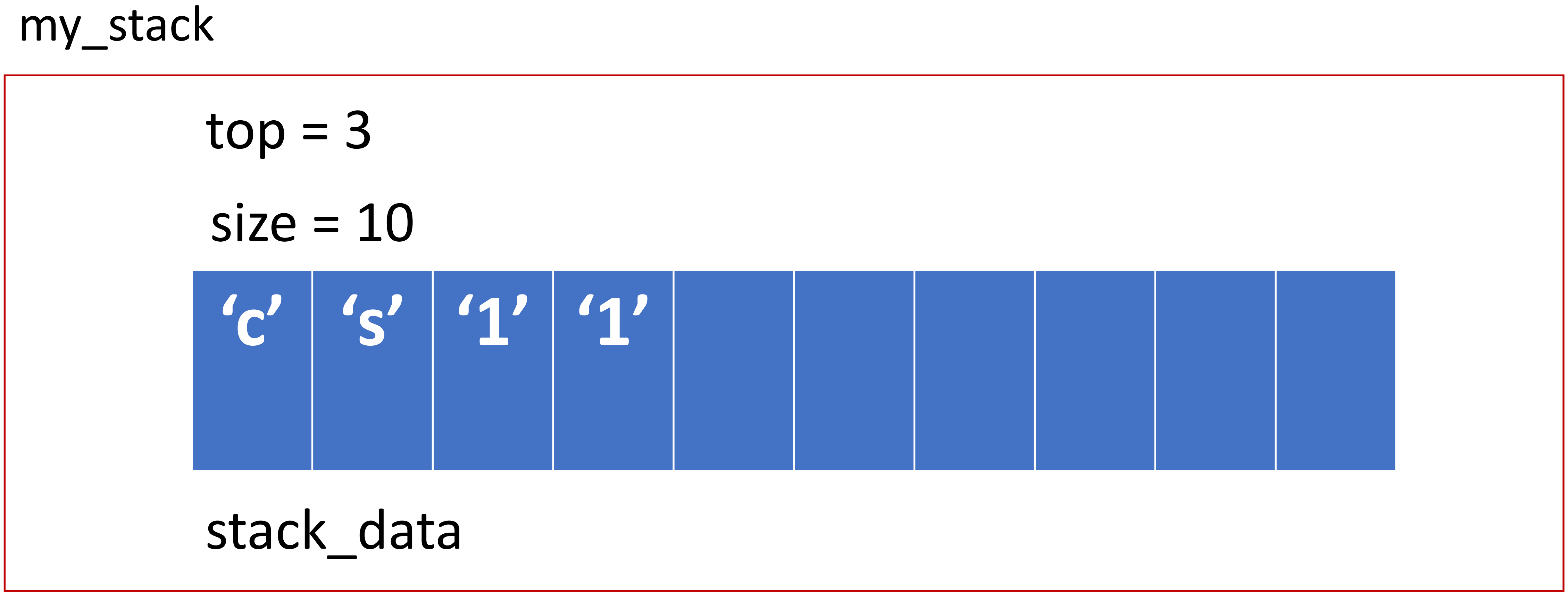
push(&my_stack, ’s’)

my_stack



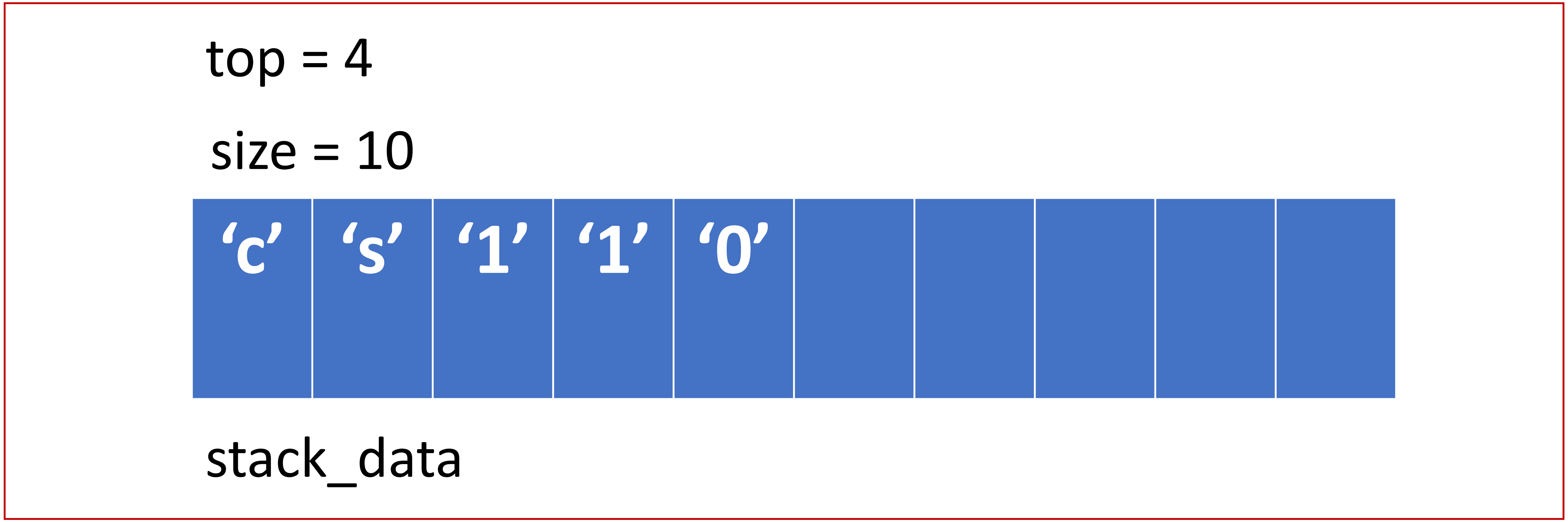
push(&my_stack, ’1’)

push(&my_stack, ’1’)



push(&my_stack, ’0’)

my_stack



Pop an item off the stack

```
char pop(struct stack *my_stack) {

    assert(my_stack->top != -1);

    char data = my_stack->stack_data[my_stack->top];

    --(my_stack->top);

    return data;
}
```

- Pop an item off of the top of the stack
  - removes an item from the stack
  - returns the value
- Precondition
  - The stack must NOT be empty
- Postcondition:
  - Stack will have one less item or be empty

Pop an item off the stack

```c
char pop(struct stack *my_stack) {
    assert(my_stack->top != -1);
    char data = my_stack->stack_data[my_stack->top];
    --(my_stack->top);
    return data;
}
```

- We can't remove an element from an empty list
- assert() check for the condition
  - terminates if the condition evaluates to false
- You can choose to print an error message and exit as well

Pop an item off the stack

```c
char pop(struct stack *my_stack) {
    assert(my_stack->top != -1);
    char data = my_stack->stack_data[my_stack->top];
    --(my_stack->top);
    return data;
}
```

- Store the current top element in variable *data*
- Decrement top and return the data
- We don’t free up the memory
  - mystack->top is keeping track of current top
  - If elements are pushed, the old value will be overwritten

Popping item from top of stack

my_stack

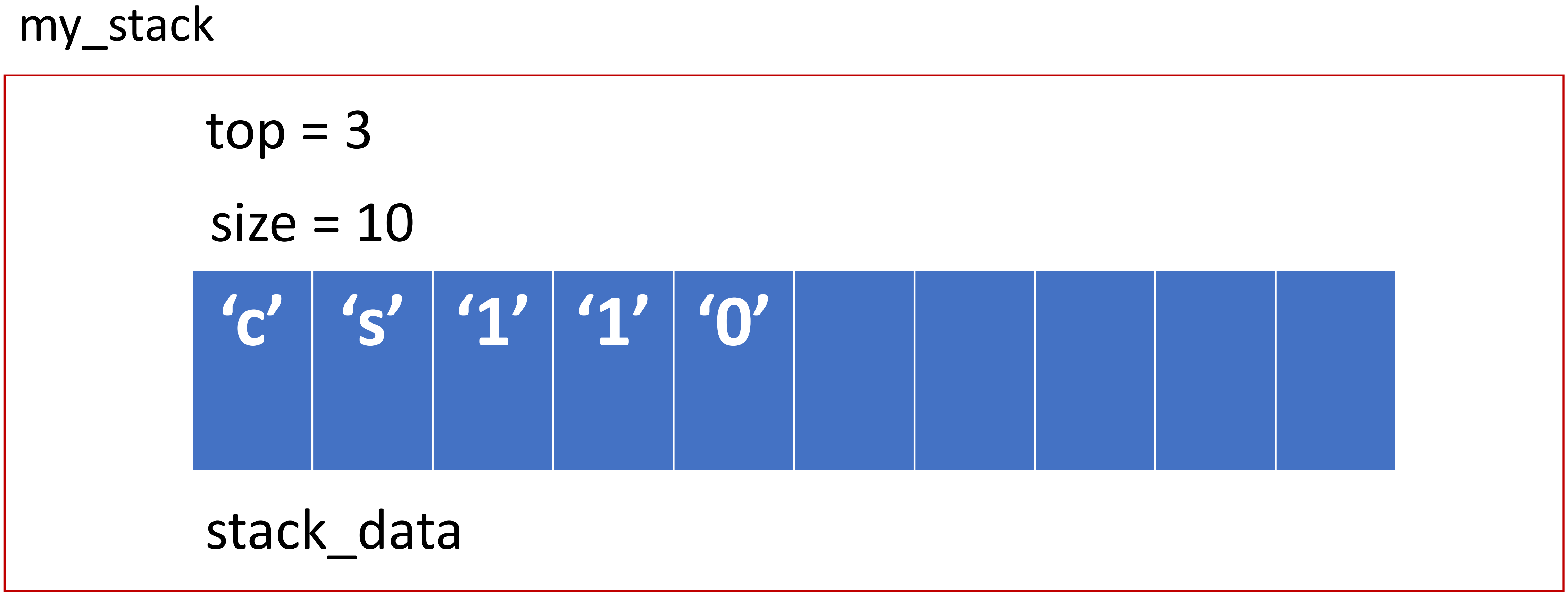
top = 4

size = 10

‘c’	‘s’	‘1’	‘1’	‘0’					

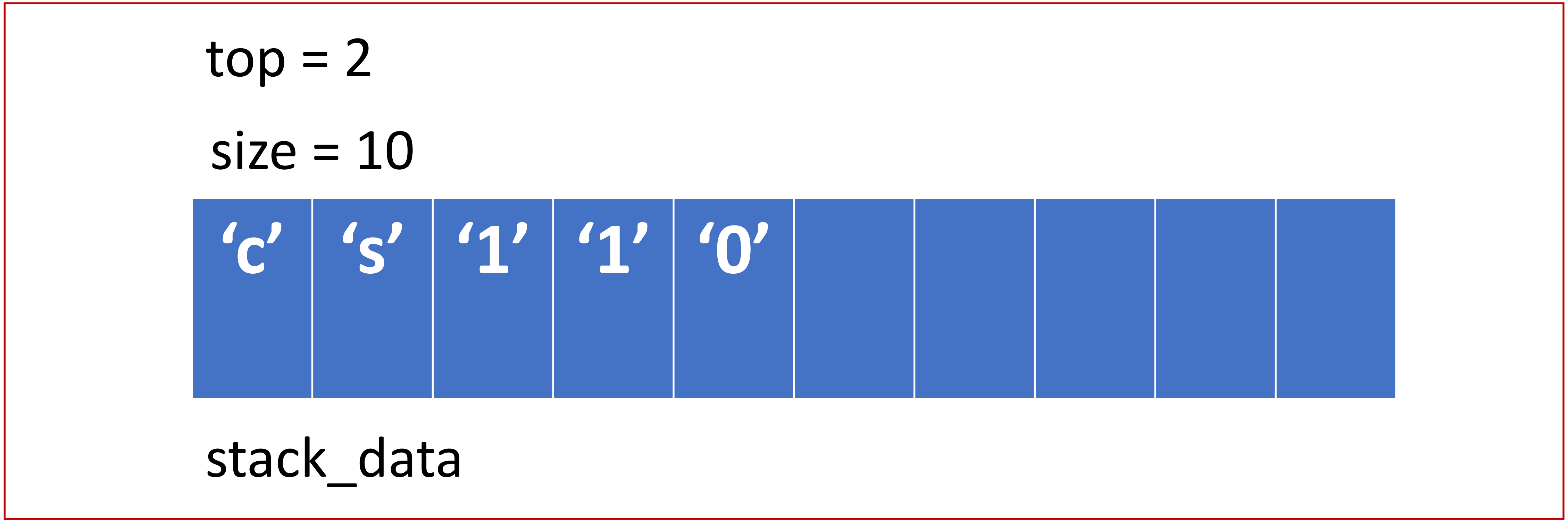
stack_data

Popping item from top of stack (return ‘0’)



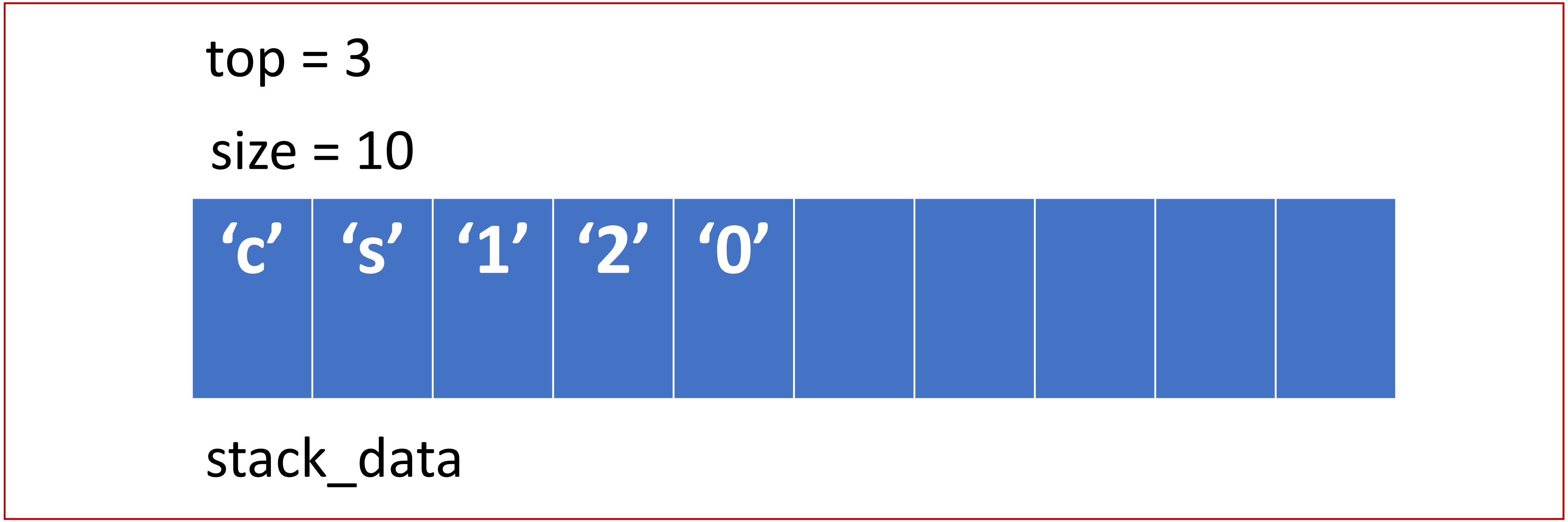
Popping item from top of stack (return ‘1’)

my_stack



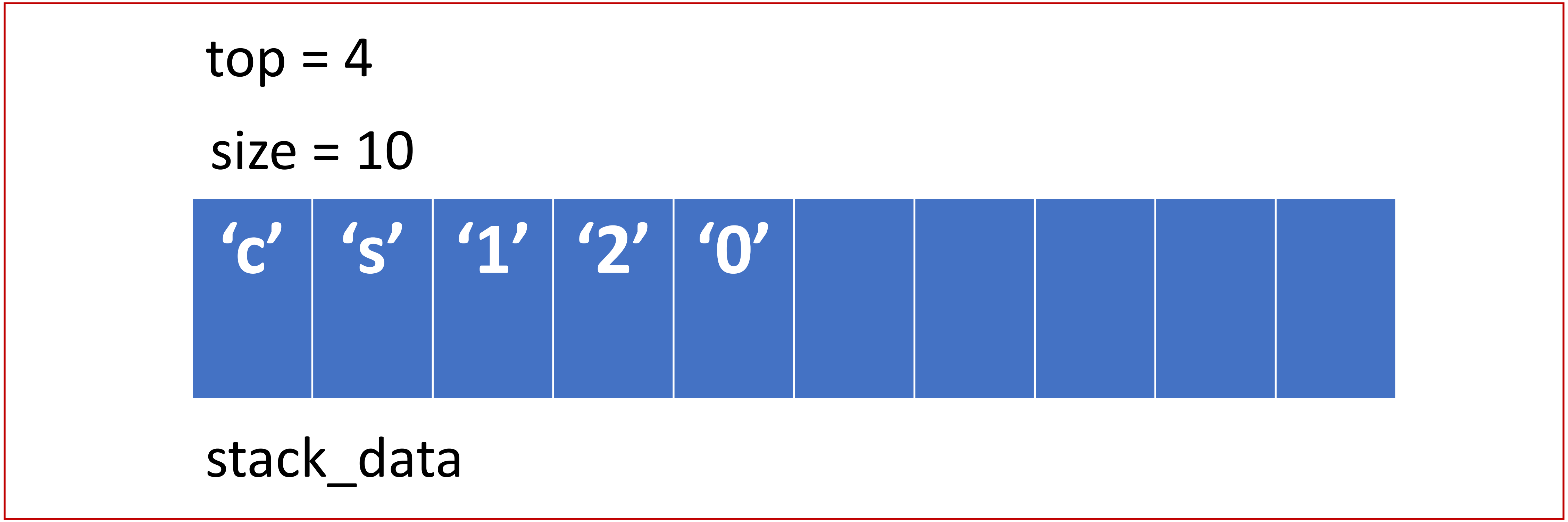
push(&my_stack, ‘2’)

my_stack



push(&my_stack, ‘0’)

my_stack



Display Items on Stack

```
void display_stack(const struct stack *my_stack) {
    if (my_stack->top == -1) {
        printf("Stack is Empty!\n");
    }
    else {
        for (size_t i = 0; i <= my_stack->top; ++i) {
            printf("%c", my_stack->stack_data[my_stack->top - i]);
            if (i == 0) {
                printf(" -> TOP");
            }
            printf("\n");
        }
    }
}
```

- If the stack is empty, there are no values to print

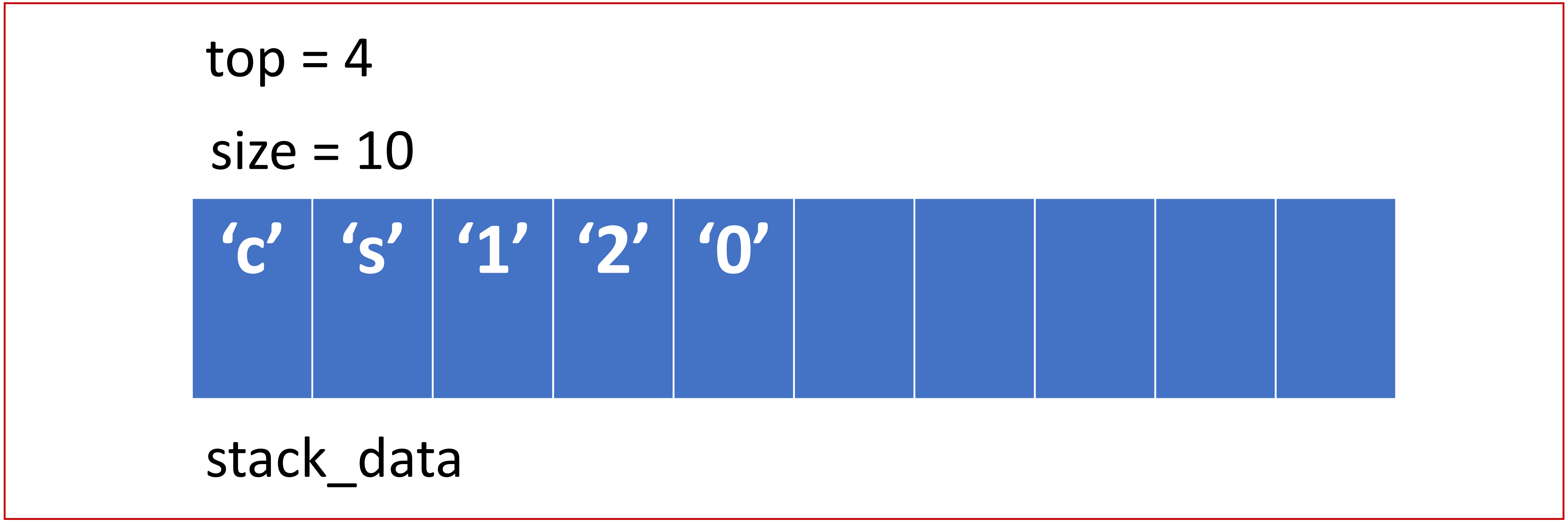
Display Items on Stack

```c
void display_stack(const struct stack *my_stack) {
    if (my_stack->top == -1) {
        printf("Stack is Empty!\n");
    }
    else {
        for (size_t i = 0; i <= my_stack->top; ++i) {
            printf("%c", my_stack->stack_data[my_stack->top - i]);
            if (i == 0) {
                printf(" -> TOP");
            }
        printf("\n");
        }
    }
}
```

- Stack is a Last In First Out data structure
- Printing elements in order that they'll be popped

display_stack(&my_stack)

my_stack



display_stack(&my_stack)

- 0 ->TOP
- 2
- 1
- s
- c

Checking if Stack is Empty

```
bool is_stack_empty(const struct stack *my_stack) {
    return my_stack->top == -1;
}
```

- Stack can be implemented in different ways
- Top can indicate the index at which the next pushed item will be stored
- In that case initial top would be 0

Freeing Up Dynamic Memory

```
void free_stack(struct stack *my_stack) {
    free(my_stack->stack_data);
}
```

- Freeing up the dynamically allocated array
- The other variables get cleaned up automatically since they're on the stack