Stacks

Last updated 6/23/23

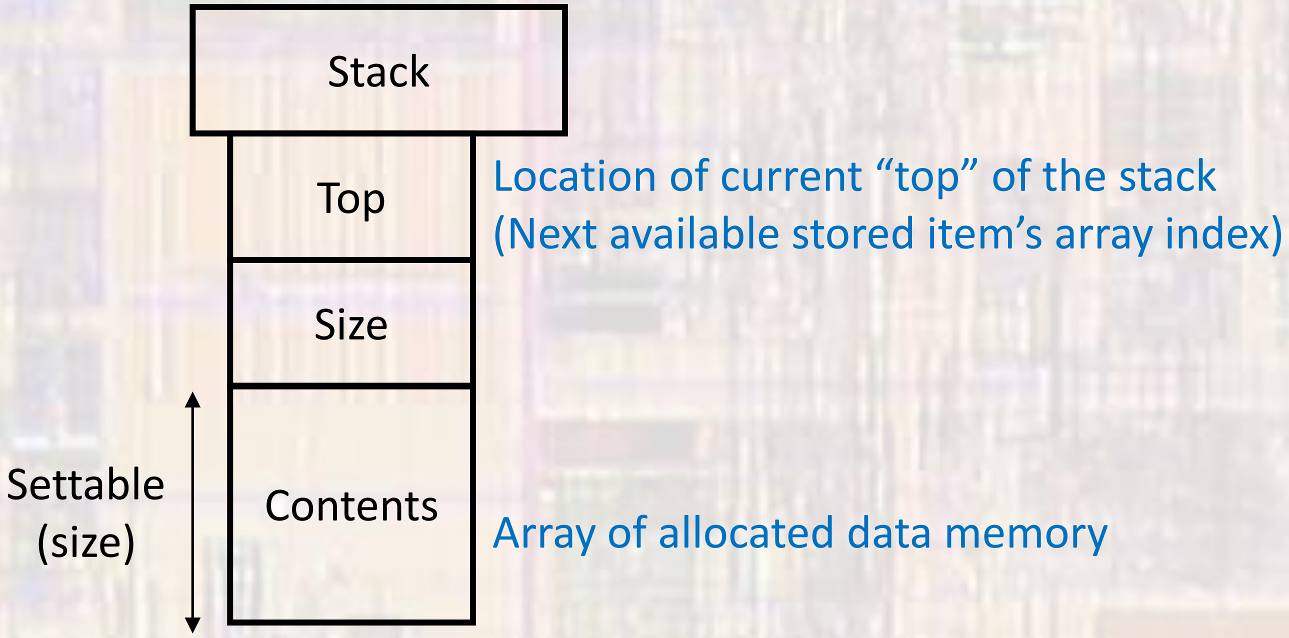
These slides introduce stacks

Stacks

- Motivation
  - We have seen how useful a stack can be in our overall computing paradigm
    - Allows reuse of limited memory
    - Easy to understand
  - Can also be used as a Last-In First-out buffer (LIFO)

Stack

- Basic Structure
  - Each Stack is a structure



Location of current "top" of the stack
(Next available stored item's array index)

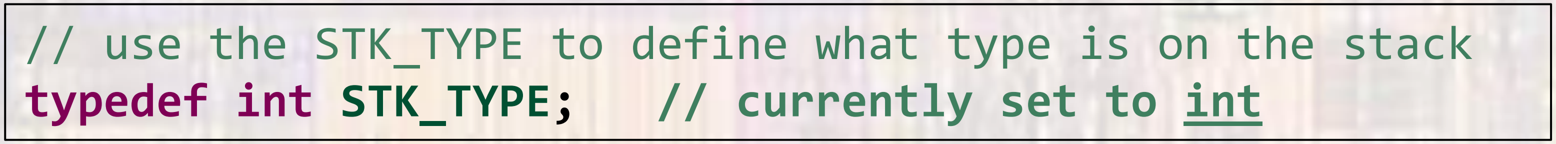
Array of allocated data memory

Stack

- Stack Functions
  - Create stack – assign name and size
  - Push – add something to the stack
  - Pop – remove something from the stack
  - Delete stack – deallocate memory
  - Stack Empty?
  - Stack Full ?
  - Empty stack – set top back to 0
    - Does not erase anything – just resets the next available spot

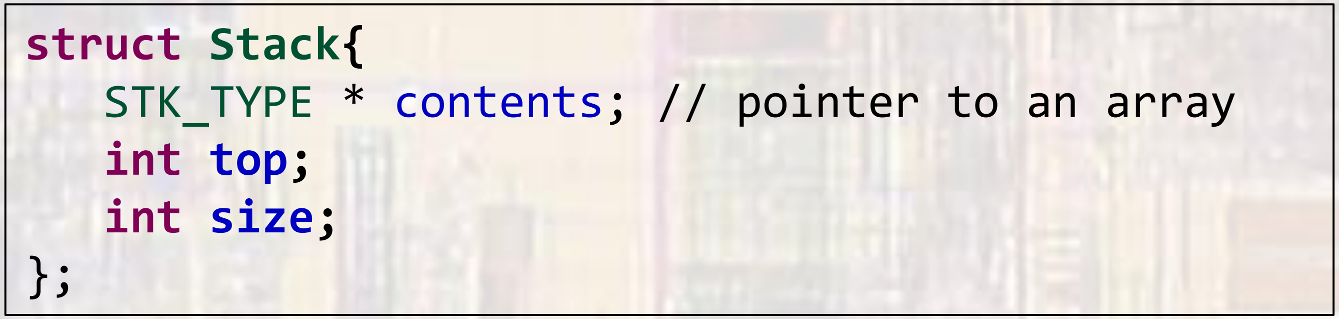
Stack

- Stack Structure
  - Our stack must store a single type of data
    - To make it easy to change the data type – create a new type that we can change in one place



```
// use the STK_TYPE to define what type is on the stack
typedef int STK_TYPE;   // currently set to int
```

  - Our stack structure needs a top, an array, and a size

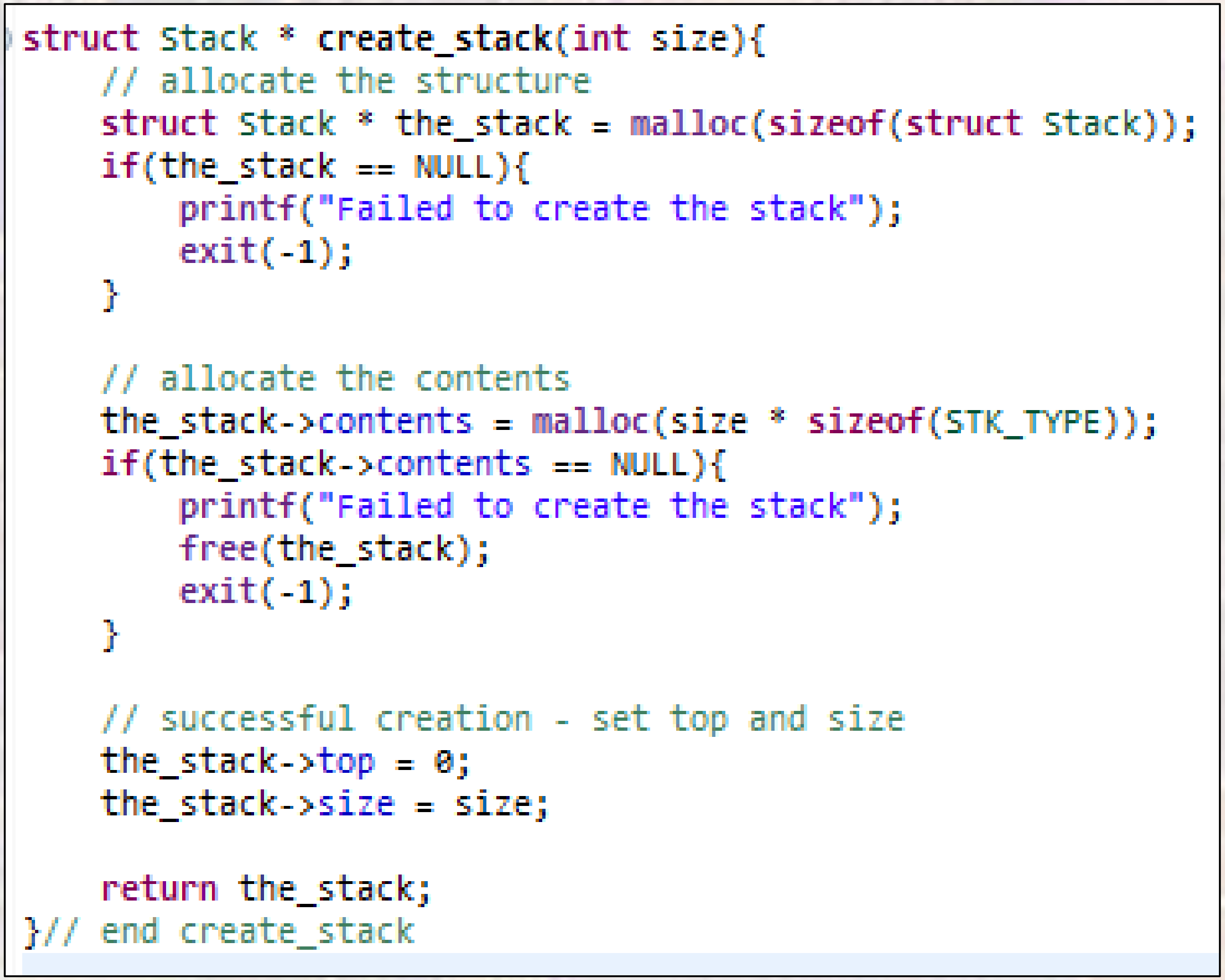


```
struct Stack{
   STK_TYPE * contents; // pointer to an array
   int top;
   int size;
};
```

Stack

- Create Stack

Note: our stack is a pointer to an allocated memory location

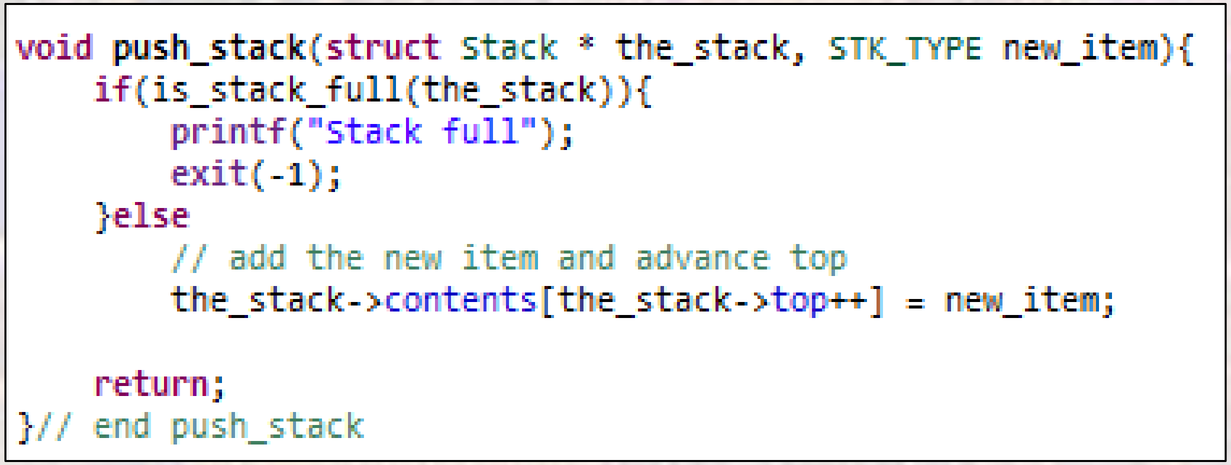


```
struct Stack * create_stack(int size){
    // allocate the structure
    struct Stack * the_stack = malloc(sizeof(struct Stack));
    if(the_stack == NULL){
        printf("Failed to create the stack");
        exit(-1);
    }

    // allocate the contents
    the_stack->contents = malloc(size * sizeof(STK_TYPE));
    if(the_stack->contents == NULL){
        printf("Failed to create the stack");
        free(the_stack);
        exit(-1);
    }

    // successful creation - set top and size
    the_stack->top = 0;
    the_stack->size = size;

    return the_stack;
}// end create_stack
```

Stack

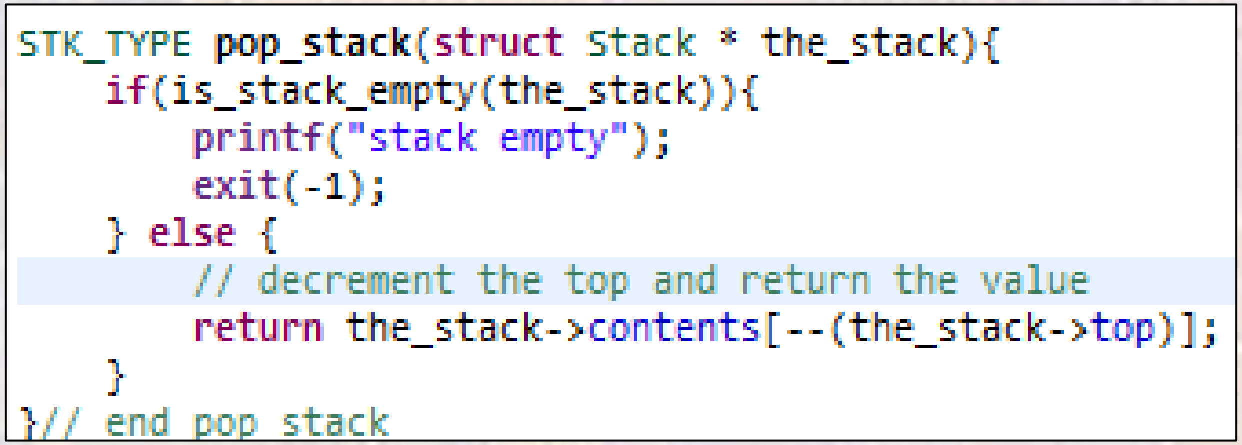
- Push Stack

Note: our stack is a pointer to an allocated memory location



```
void push_stack(struct Stack * the_stack, STK_TYPE new_item){
    if(is_stack_full(the_stack)){
        printf("Stack full");
        exit(-1);
    }else
        // add the new item and advance top
        the_stack->contents[the_stack->top++] = new_item;

    return;
}// end push_stack
```

- Pop Stack

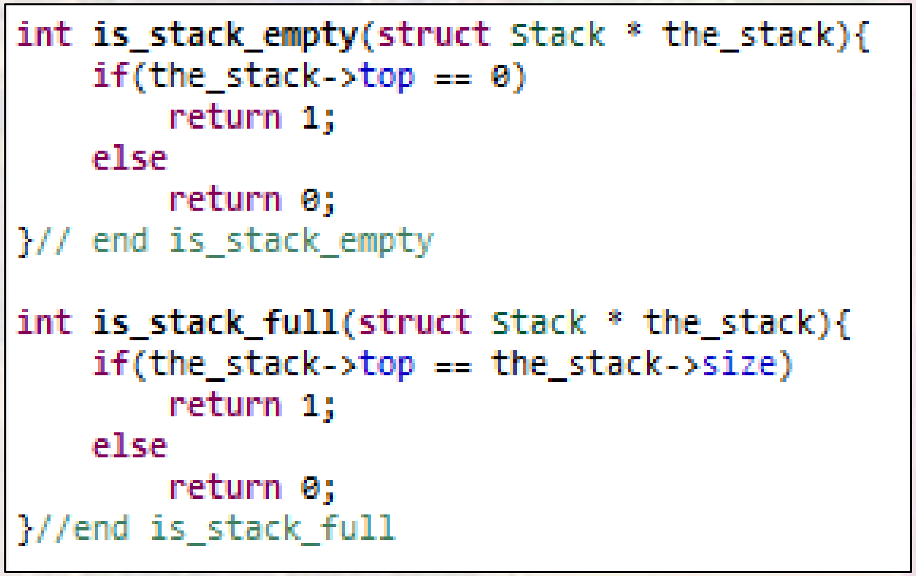


```
STK_TYPE pop_stack(struct Stack * the_stack){
    if(is_stack_empty(the_stack)){
        printf("stack empty");
        exit(-1);
    } else {
        // decrement the top and return the value
        return the_stack->contents[--(the_stack->top)];
    }
}// end pop_stack
```

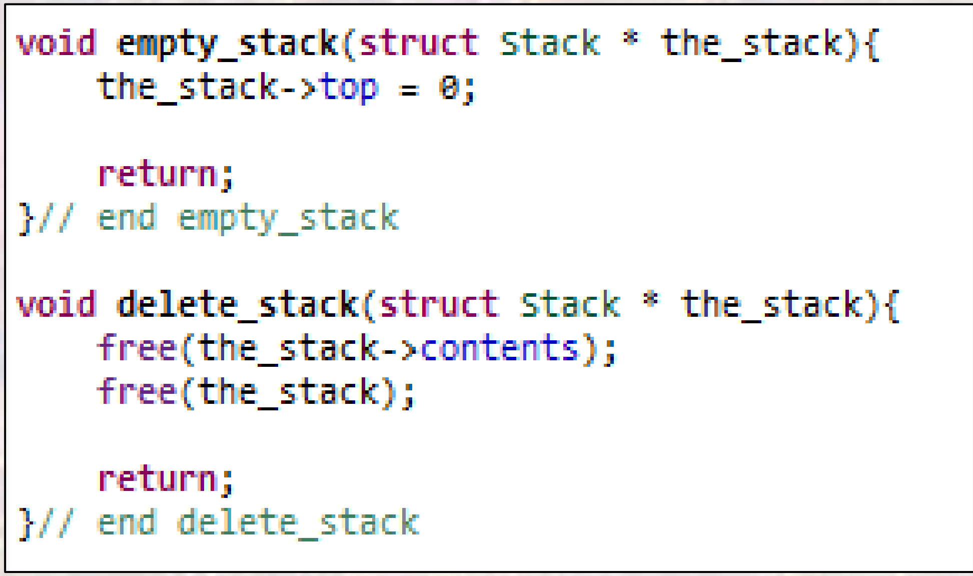
Stack

- Helper Functions

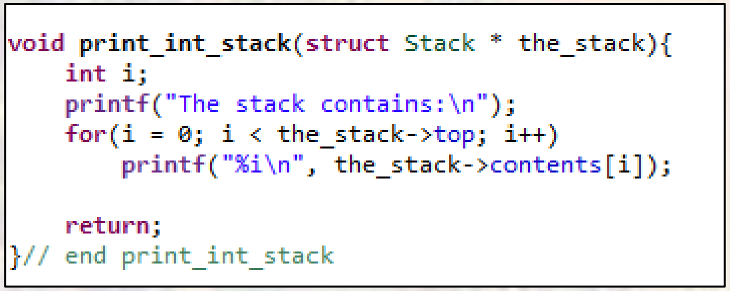
Note: our stack is a pointer to an allocated memory location



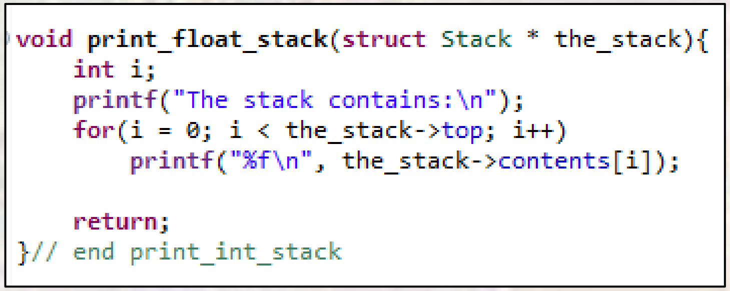
```
int is_stack_empty(struct Stack * the_stack){
    if(the_stack->top == 0)
        return 1;
    else
        return 0;
}// end is_stack_empty

int is_stack_full(struct Stack * the_stack){
    if(the_stack->top == the_stack->size)
        return 1;
    else
        return 0;
}//end is_stack_full
```



```
void empty_stack(struct Stack * the_stack){
    the_stack->top = 0;

    return;
}// end empty_stack

void delete_stack(struct Stack * the_stack){
    free(the_stack->contents);
    free(the_stack);

    return;
}// end delete_stack
```



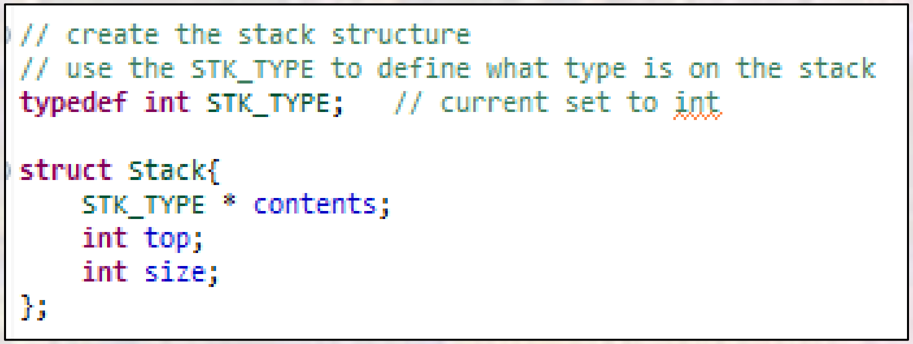
```
void print_int_stack(struct Stack * the_stack){
    int i;
    printf("The stack contains:\n");
    for(i = 0; i < the_stack->top; i++)
        printf("%i\n", the_stack->contents[i]);

    return;
}// end print_int_stack
```



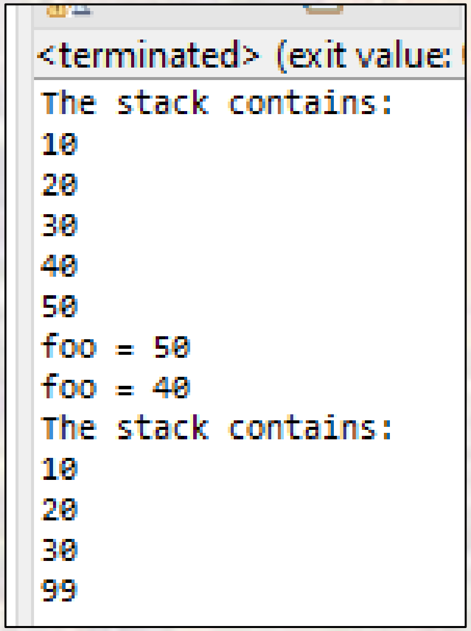
```
void print_float_stack(struct Stack * the_stack){
    int i;
    printf("The stack contains:\n");
    for(i = 0; i < the_stack->top; i++)
        printf("%f\n", the_stack->contents[i]);

    return;
}// end print_int_stack
```

Stack

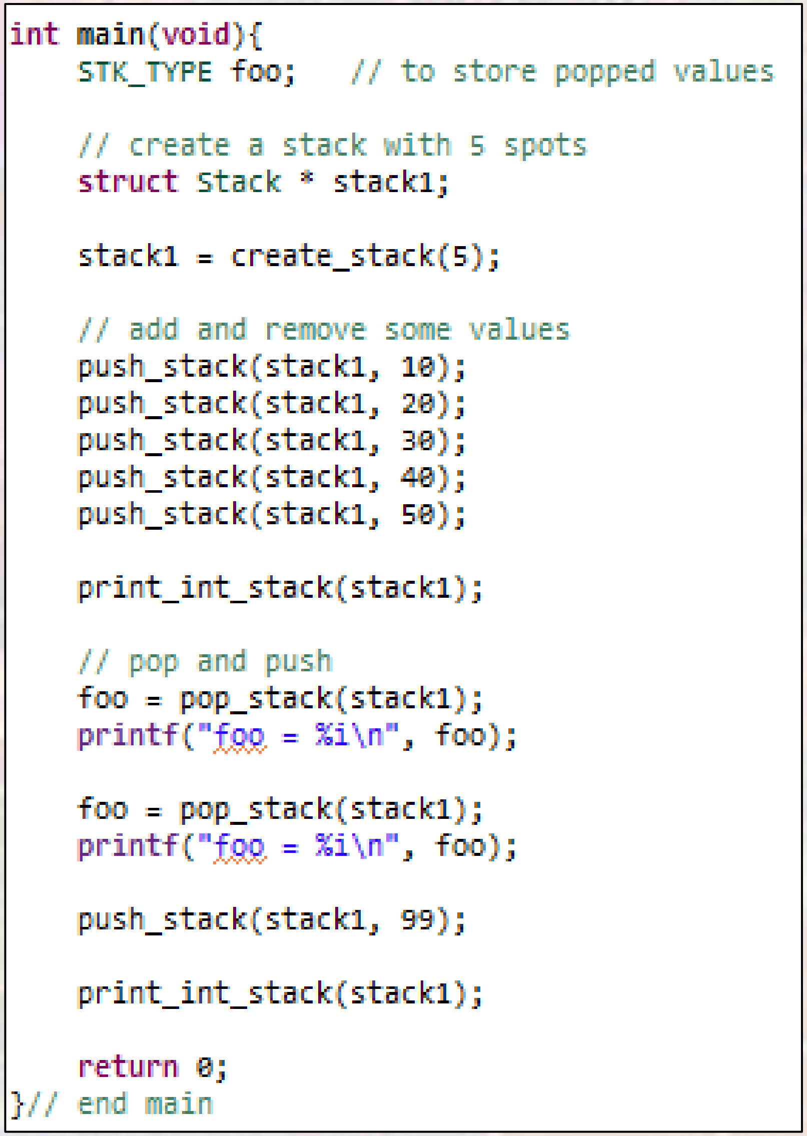
- Example - int



```
// create the stack structure
// use the STK_TYPE to define what type is on the stack
typedef int STK_TYPE;   // current set to int

struct Stack{
    STK_TYPE * contents;
    int top;
    int size;
};
```



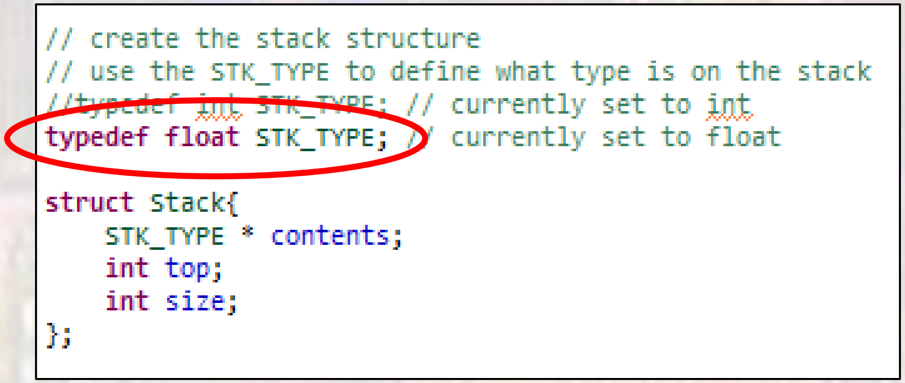
```
The stack contains:
10
20
30
40
50
foo = 50
foo = 40
The stack contains:
10
20
30
99
```



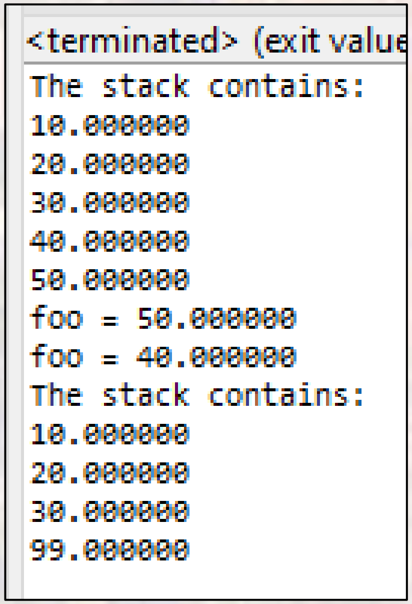
```
int main(void){
    STK_TYPE foo;   // to store popped values

    // create a stack with 5 spots
    struct Stack * stack1;

    stack1 = create_stack(5);

    // add and remove some values
    push_stack(stack1, 10);
    push_stack(stack1, 20);
    push_stack(stack1, 30);
    push_stack(stack1, 40);
    push_stack(stack1, 50);

    print_int_stack(stack1);

    // pop and push
    foo = pop_stack(stack1);
    printf("foo = %i\n", foo);

    foo = pop_stack(stack1);
    printf("foo = %i\n", foo);

    push_stack(stack1, 99);

    print_int_stack(stack1);

    return 0;
}// end main
```

Stack

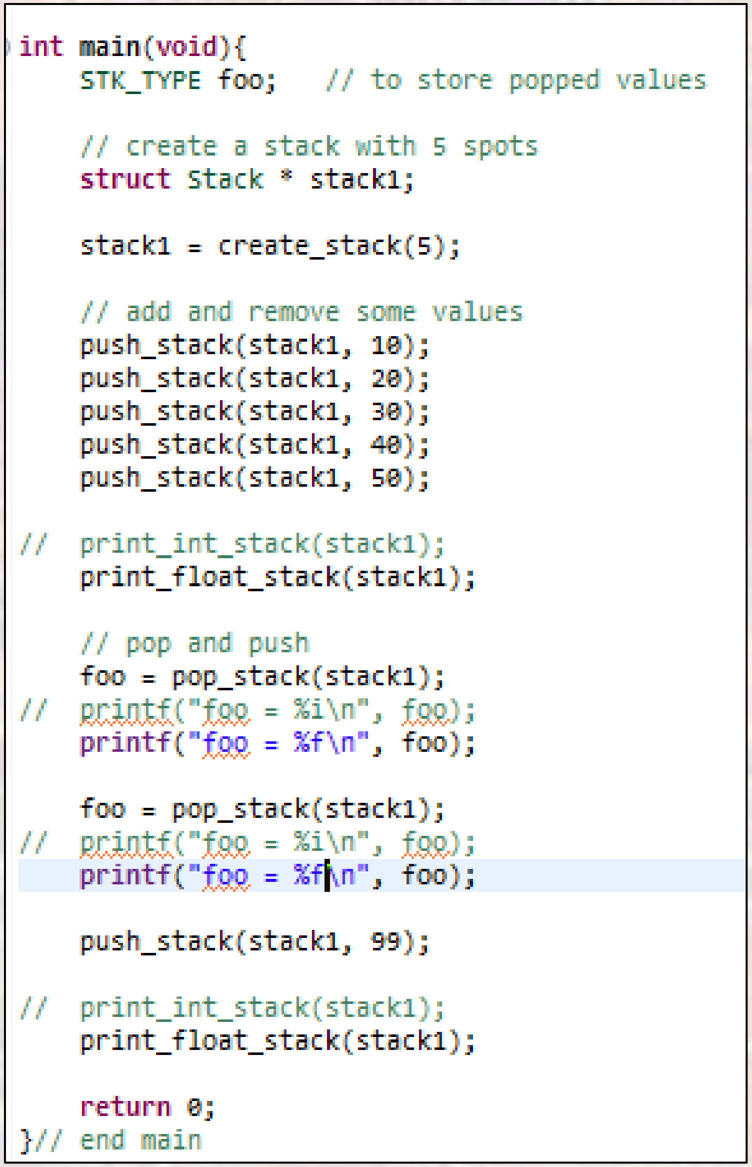
- Example – float
  - No changes to stack functions



```
// create the stack structure
// use the STK_TYPE to define what type is on the stack
//typedef int STK_TYPE; // currently set to int
typedef float STK_TYPE; // currently set to float

struct Stack{
    STK_TYPE * contents;
    int top;
    int size;
};
```



```
<terminated> (exit value
The stack contains:
10.000000
20.000000
30.000000
40.000000
50.000000
foo = 50.000000
foo = 40.000000
The stack contains:
10.000000
20.000000
30.000000
99.000000
```



```
int main(void){
    STK_TYPE foo;   // to store popped values

    // create a stack with 5 spots
    struct Stack * stack1;

    stack1 = create_stack(5);

    // add and remove some values
    push_stack(stack1, 10);
    push_stack(stack1, 20);
    push_stack(stack1, 30);
    push_stack(stack1, 40);
    push_stack(stack1, 50);

//  print_int_stack(stack1);
    print_float_stack(stack1);

    // pop and push
    foo = pop_stack(stack1);
//  printf("foo = %i\n", foo);
    printf("foo = %f\n", foo);

    foo = pop_stack(stack1);
//  printf("foo = %i\n", foo);
    printf("foo = %f\n", foo);

    push_stack(stack1, 99);

//  print_int_stack(stack1);
    print_float_stack(stack1);

    return 0;
}// end main
```