Stack Exercises

Example: Write a program that reads a sentence, ending with a full-stop, and outputs its characters in reverse order.

```
int main(void)
{
    char ch;
    stack_t sentence;
    initialize_s(&sentence);
    // Read the sentence and push into a stack
    printf("Enter a sentence ending with a full-stop:\n");
    ch = getchar();
    while (ch != '.')
    {
        push(&sentence, ch);
        ch = getchar();
    }
    // Repeat until the stack becomes
    empty while (!is_empty_s(&sentence))
        putchar(pop(&sentence));
    printf("\n");
    return (0);
}
```

- Notice that '.' is not pushed onto the stack, and hence it is not displayed. If you want it to be displayed, you should call `push` once more between two while loops.

Home Exercise: Given a sentence, ending with a question mark, output each word in the sentence in reverse order. For instance, if the input is

     WHERE ARE YOU?

the output should be

     EREHW ERA UOY?

Example: Trace the program below, and find its output:
(Remember, `STACK_SIZE` is declared as 20!)

```
int main(void)
{
    stack_t stack;
    char item, ch;
    int i;
    initialize_s(&stack);
    item = pop(&stack);
    for (ch = 'a'; ch <= 'z'; ch++)
        push(&stack, ch);
    for (i = 0; i < STACK_SIZE; i++)
    {
        item = pop(&stack);
        printf("%c", item);
    }
    return (0);
}
```

Output:

Error: Stack is empty!
Error: Stack is full!
Error: Stack is full!
Error: Stack is full!
Error: Stack is full!
Error: Stack is full!
Error: Stack is full!
tsrqponmlkjihgfedcba

Example: Given an arithmetic expression, determine whether the parentheses are balanced or not. For example:

a + (b / (c + d) + (a - b)   a + b / (c + d)) + (a - b)

                  not closed   not opened

- Can we solve this problem by counting the opening and closing parentheses and comparing them? No, because then, in the following expressions, the parentheses would be found as balanced, although they are not:

    a) + (b / (c + d) + (a - b)   a + b / (c + d)) + ((a - b)

- In an arithmetic expression, the first closing parenthesis encountered is supposed to match the last opening parenthesis. For instance, in the following expression:

a + b / (c + d)) + ((a - b)

the first closing parenthesis matchs the first opening parenthesis, and the second closing parenthesis does not have a corresponding opening parenthesis. Thus, the parentheses are not balanced.

- Using this logic, we will read the given expression into a string. Then, we will go over it character by character, pushing each opening parenthesis onto the stack, and popping the stack each time a closing parenthesis is found.

- When we find a closing parenthesis, if we can not pop the stack because the stack is already empty, this means that there is an extra closing paranthesis. We can display a proper message and stop the operation, since we decide that the parentheses are not balanced.

- Otherwise, the operation should go on until we reach to the end of the expression. When the expression is finished, if the stack is not empty, this means that there is a missing closing paranthesis.

- The parentheses are balanced only if we have been able to reach to the end of the expression, pushing and popping when necessary without error, and the stack becomes empty at the end.

```c
void check_par(char *exp) {
    stack_t stack;
    int k = 0, flag = 1; //initially no
    error char x;
    int size = strlen(exp);
    initialize_s(&stack);
    while (k < size && flag) {
        if (exp[k] == '(')
            push(&stack, '(');
        else if (exp[k] == ')')
            if (!is_empty_s(&stack))
                x = pop(&stack);
            else {
                printf("Missing (\n");
                flag = 0; //found error, exit loop
            }
        k++;
    }
    if (!is_empty_s(&stack))
        printf("Missing )\n");
    else if (flag) //no error, loop finished
        normally printf("Balanced\n");
}
```

Home Exercise: Rewrite the above function without using a stack.

- Upto now, we always used character stacks in our examples, because we defined the functions for stack operations for character stacks. Modifying those functions for different type of stacks is not difficult. They are modified for integer stacks and put into the header file `stack_int.h`.

Example: Define a function that reverses the elements of a given integer array. For instance, if the given array contains 2, 4, 3, and 5, after the function call, it should contain 5, 3, 4, and 2.

- If we push the array elements onto a stack, and then pop them back, they will be reversed.

```
#include <stack_int.h>

void reverse(int a[], int size)
{
    int k;
    stack_t stack;
    initialize_s(&stack);
    for (k = 0; k < size; k++)
        push(&stack, a[k]);
    for (k = 0; k < size; k++)
        a[k] = pop(&stack);
}
```

Example: Display the binary equivalent of a decimal number using a stack.

```
void binary_eq(int num)
{
    stack_t stack;
    initialize_s(&stack);
    while (num != 0)
    {
        push(&stack, num % 2);
        num = num / 2;
    }
    while (!is_empty_s(&stack))
        printf("%d", pop(&stack));
}
```

Home Exercise: Modify the above function so that it does not display but returns the binary equivalent of a decimal number within an array.

Assume that we declared an integer stack `s`, and it already contains the following data:

```
s.data
```

```
   4 |  34| s.top = 4
   3 |   6|
   2 |  12|
   1 |   7|
   0 |  25|
```
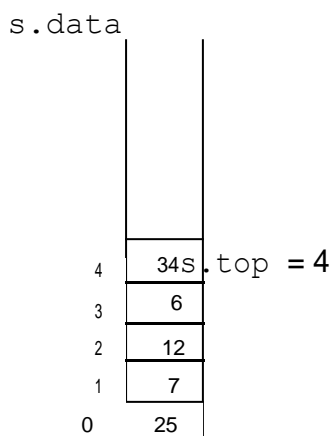
- Let's write program segments to solve the following problems, using the stack `s`.

- **Problem:** Remove the top 3 items (means we want to remove 34, 6 and 12).

```
for(k = 1; k <= 3; k++)
    num = pop(&s);
```

- **Problem:** Remove only the third item from the top (means we want to remove 12, but we want to keep the others).

- As you know, we can not reach 12 without first popping 34 and 6. However, we want to put 34 and 6 back into the stack. In this case, we need to store 34 and 6 into somewhere. We will use an extra stack to store those data.

```
stack_t e;
/* initialize e as an empty stack
*/ initialize_s(&e);
/* pop the first two items from s, and push them onto e
*/ for (k = 1; k < 3; k++)
    {num = pop(&s);
     push(&e, num);
}
/* remove the third item */
num = pop(&s);
/* pop the items from e, and push them onto s
*/ for (k = 1; k < 3; k++)
    push(&s, pop(&e));
```

Home Exercise: Write a function that removes and returns the nth item from the top of a stack, returns -9999 if the stack has less than n items.

- **Problem:** Remove the bottom item.

```
/* initialize e as an empty stack
*/ initialize_s(&e);
/* pop all items from s, and push them onto e
*/ while (!is_empty_s(&s))
    push(&e, pop(&s));
/* remove the top item from e,
   because it is the bottom item of s */
num = pop(&e);
/* pop the remaining items from e, and push them onto s
*/ while (!is_empty_s(&e))
    push(&s, pop(&e));
```

Home Exercise: Write a function that removes and returns the bottom item of a stack.

- **Problem:** Remove 7, but do not lose the other data items.

```
/* initialize e as an empty stack
*/ initialize_s(&e);

/* repeat until s becomes empty */
while (!is_empty_s(&s))
{
    /* remove an item from s */
    num = pop(&s);
    /* push it onto e only if it is not 7
    */ if (num != 7)
        push(&e, num);
}

/* pop all items from e, and push them onto s
*/ while (!is_empty_s(&e))
    push(&s, pop(&e));
```

Home Exercise: Write a function that removes all occurrences of a certain item from a stack.

Example: Write a function that removes and returns the maximum element of a stack.

```c
int remove_max(stack_t *s)
{
    stack_t e;
    int num, max;
    initialize_s(&e);
    max = pop(s);
    push(&e, max);
    while (!is_empty_s(s)) {
        num = pop(s);
        push(&e, num);
        if (num > max)
            max = num;
    }
    while (!is_empty_s(&e)) {
        num = pop(&e);
        if (num != max)
            push(s, num);
    }
    return (max);
}
```

Example: Write a function that moves the nth element from the top to the bottom of a stack.

```c
void move_nth_bottom (stack_t *s, int n)
    { stack_t e;
    int k, nth;
    initialize_s(&e);

    for (k = 1; k < n; k++)
        push(&e, pop(s));
    /* remove the nth item */
    nth = pop(s);
    /* remove the rest */
    while (!is_empty_s(s))
        push(&e, pop(s));
    /* push nth to the bottom */
    push(s, nth);
    while (!is_empty_s(&e))
        push(s, pop(&e));
}
```

- What about if there are less than `n` elements in the stack? We may check it at the beginning of the function, because top of the stack tells us the number of elements in the stack (`s->top + 1`). We also don't need to change the stack if the nth element is the bottom element. Thus, we can take the whole function body into the following if statement:

```
if (n <= s->top) {
...
}
```

- Stack structure is mainly used by system programs such as operating systems and compilers. For instance, compilers implement function calls making use of stacks.

- Consider the following program skeleton containing three function definitions:
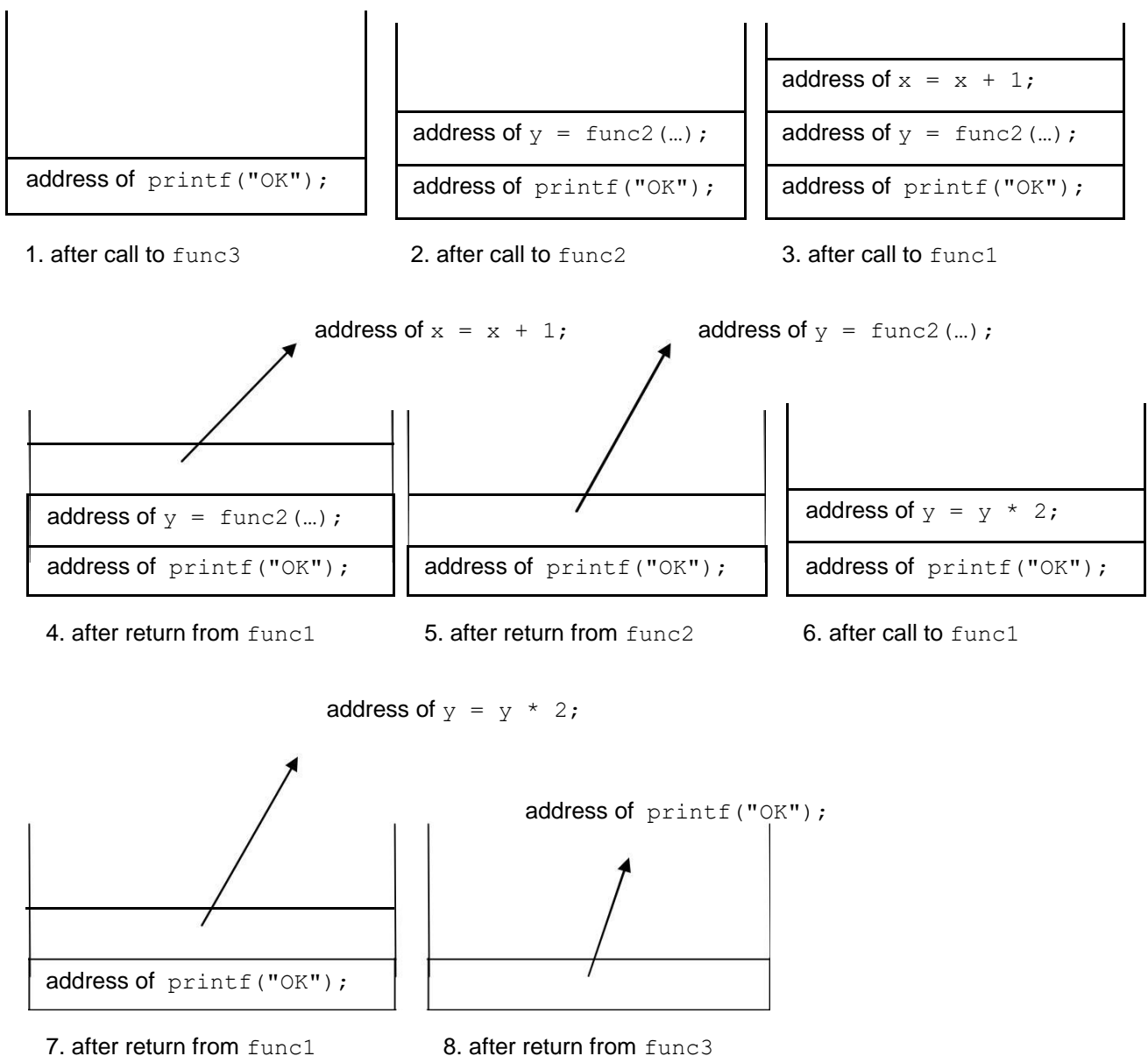
```
void func1(...)
{
     ...
}

int func2(...)
{
     ...
     func1(...);
     x = x + 1;
     return (x);
}

void func3(...)
{
     ...
     y = func2(...);
     func1(...);
     y = y * 2;
     ...
}

int main(void)
{
     ...
     func3(...);
     printf("OK");
     ...
}
```
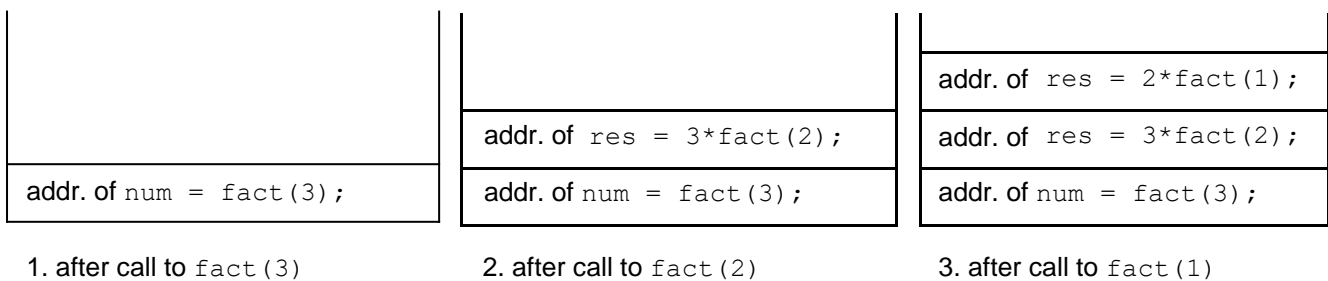
- Given the program skeleton, examine the order of the operations.

- Each time a function is called, the computer must remember where to return after completion of that function.

- The compilers use a stack for that purpose. Each time a function call is made, the return address, i.e., the address of the next instruction, is pushed onto a stack. Each time a function is completed, the top item on the stack is popped to determine the memory address to which the return operation should be made.

- Therefore, when a compiler tries to implement the above program, the stack will look like as follows:

		address of `x = x + 1;`
	address of `y = func2(…);`	address of `y = func2(…);`
address of `printf("OK");`	address of `printf("OK");`	address of `printf("OK");`

1. after call to `func3`   2. after call to `func2`   3. after call to `func1`

address of `x = x + 1;`   address of `y = func2(…);`

		address of `y = y * 2;`
address of `y = func2(…);`		address of `y = y * 2;`
address of `printf("OK");`	address of `printf("OK");`	address of `printf("OK");`

4. after return from `func1`   5. after return from `func2`   6. after call to `func1`

address of `y = y * 2;`

address of `printf("OK");`

address of `printf("OK");`

7. after return from `func1`   8. after return from `func3`

- The compilers use the same logic to handle recursion. Consider the following recursive factorial function:

```
double fact(int n)
{
    double res;
    if (n <= 1)
        res = 1;
    else
        res = n * fact(n - 1);
    return (res);
}
```

- Assume that, the function is called from main as

    follows: `num = fact(3);`

- When a compiler tries to implement the above function call, the stack will look like as follows:

		addr. of `res = 2*fact(1);`
	addr. of `res = 3*fact(2);`	addr. of `res = 3*fact(2);`
addr. of `num = fact(3);`	addr. of `num = fact(3);`	addr. of `num = fact(3);`

1. after call to `fact(3)`   2. after call to `fact(2)`   3. after call to `fact(1)`

RES

addr. of `res = 2*fact(1);` ⟹ `res = 2 * 1 = 2`

addr. of `res = 3*fact(2);` ⟹ `res = 3 * 2 = 6`

addr. of `num = fact(3);`

| addr. of `res = 3*fact(2);` |
| addr. of `num = fact(3);` |

| |
| addr. of `num = fact(3);` |

4. after returning 1 from `fact(1)`   5. after returning 2 from `fact(2)`   6. after returning 6 from `fact(3)`