1. What can you learn from the following program?

```c
#include <stdio.h>

void test_fun(int a, void* b, char *c)
{
    *(int*)a = 11;
    *(int*)b = 22;
    *(int*)c = 33;

    return;
}

int main(void)
{
    int a = 1, b = 2, c = 3;

    test_fun(&a, &b, &c);
    printf("a=%d, b=%d, c=%d\n", a, b, c);    // a=11, b=22, c=33

    return 0;
}
```

We have learned several concept from this program:

a. In 32-bit systems, int and pointer are both 4 bytes, so accidentally treating an integer as a pointer may still work.
b. In 64-bit systems, int is 4 bytes, while a pointer is 8 bytes.
c. When we pass an address (&a) but receive it as an int, the upper 4 bytes of the pointer get lost in a 64-bit system, making it an invalid pointer.
d. This causes a segmentation fault (SIGSEGV) or undefined behavior.
e. Always use the correct data type (int* instead of int for addresses).
f. Function argument mismatches can lead to truncated pointers and undefined behavior.
g. 32-bit and 64-bit systems handle pointers differently—be mindful of size differences.
h. Always use the correct data type (int* instead of int for addresses).
i. Function argument mismatches can lead to truncated pointers and undefined behavior.
j. 32-bit and 64-bit systems handle pointers differently—be mindful of size differences

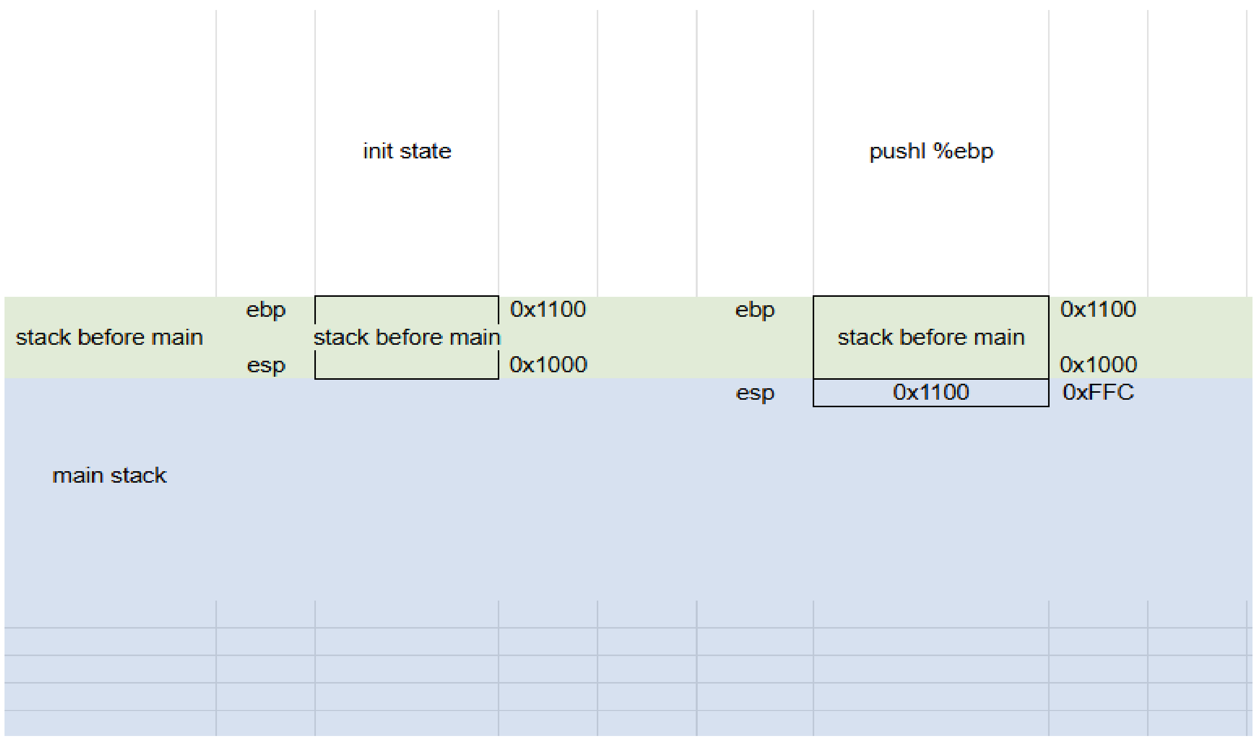
2. There is a program in file "Pointer and Function.c", Answer the following questions

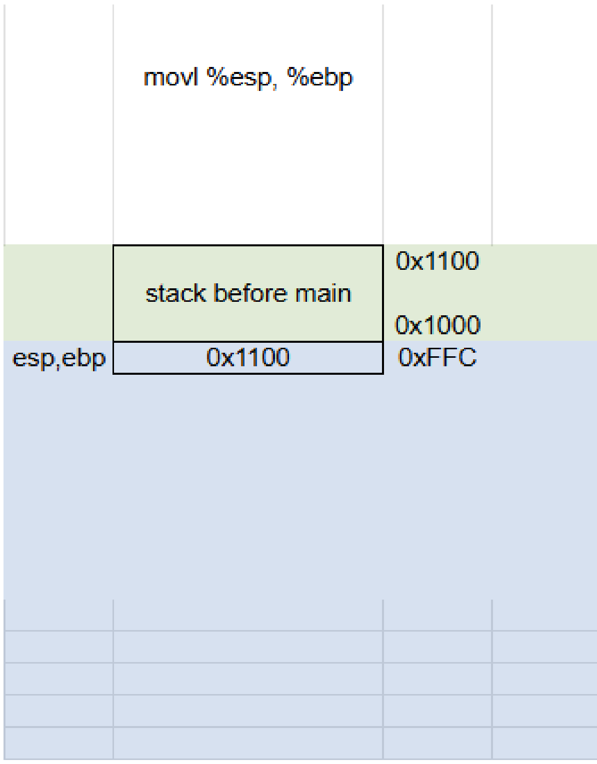
   a. Write the corresponding C code next to the assembly statement

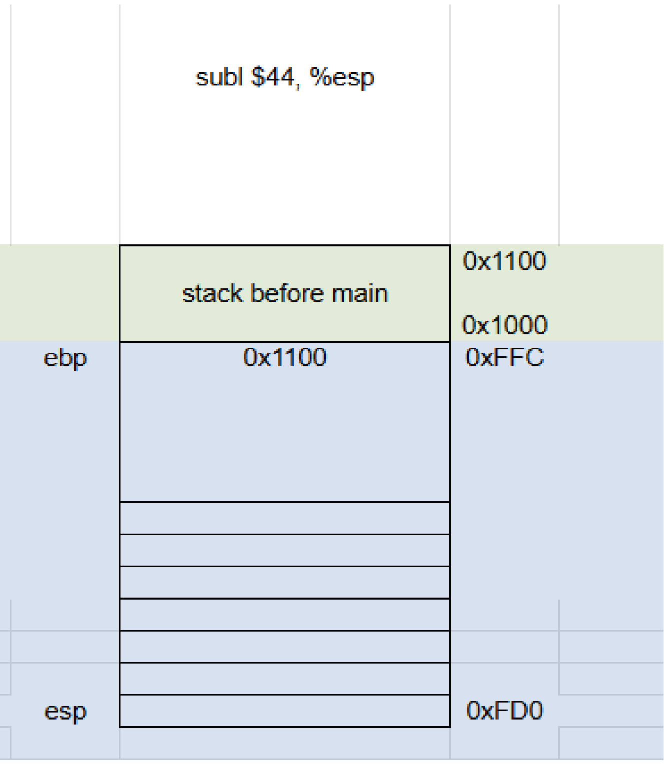
```
test_func:
        pushl   %ebp
        movl    %esp, %ebp

        addl    $6, 8(%ebp)          ; v += 6;

        movl    12(%ebp), %eax       ; *p += 7;
        leal    7(%eax), %edx
        movl    12(%ebp), %eax
        movl    %edx, (%eax)

        movl    16(%ebp), %eax       ; q[1] += 8;
        addl    $4, %eax
        movl    16(%ebp), %edx
        addl    $4, %edx
        movl    (%edx), %edx
        addl    $8, %edx
        movl    %edx, (%eax)
        nop
        popl    %ebp
        ret
```

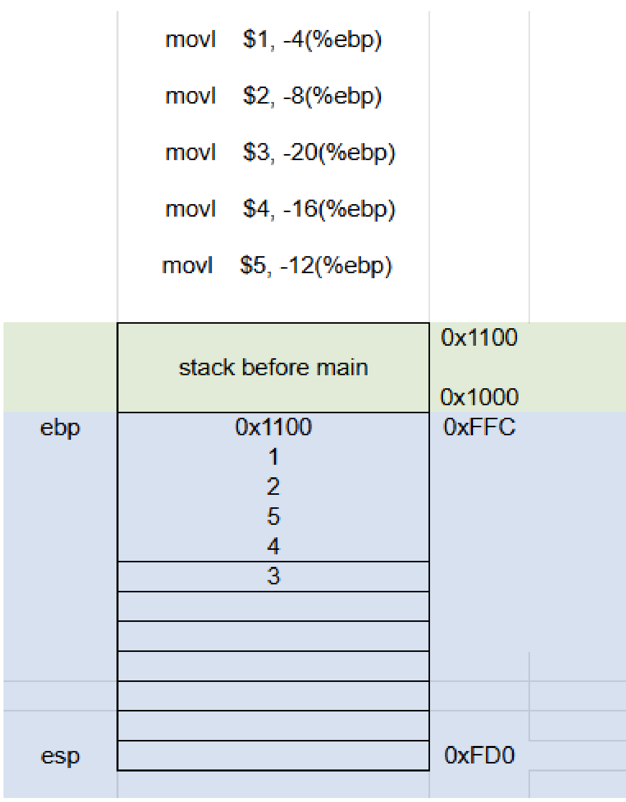
```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $44, %esp

        movl    $1, -4(%ebp)        ; a = 1;
        movl    $2, -8(%ebp)        ; b = 2;
        movl    $3, -20(%ebp)       ; c[0] = 3;
        movl    $4, -16(%ebp)       ; c[1] = 4;
        movl    $5, -12(%ebp)       ; c[2] = 5;

        leal    -20(%ebp), %eax     ; test_func(a, &b, c);
        movl    %eax, 8(%esp)
        leal    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    test_func
        movl    $0, %eax
        leave
        ret
```

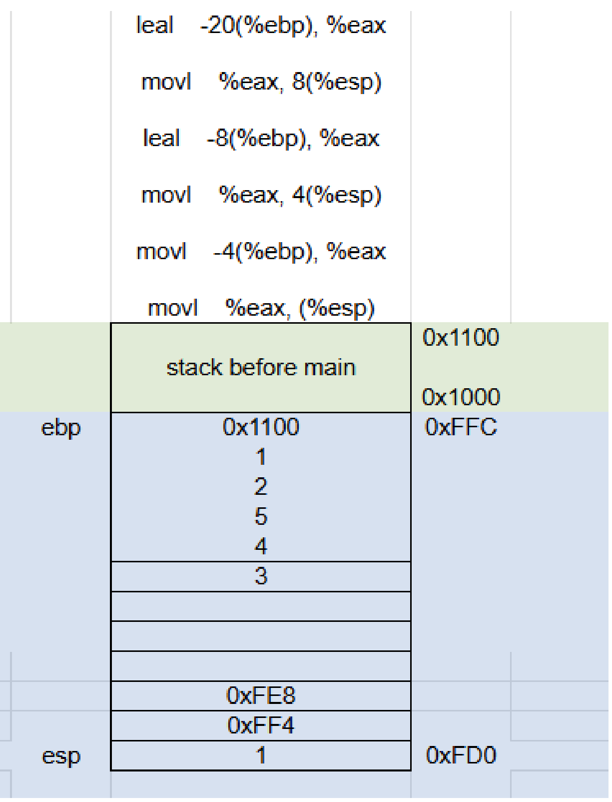
b. Draw some pictures to show the stack changes after each assembly instruction is executed

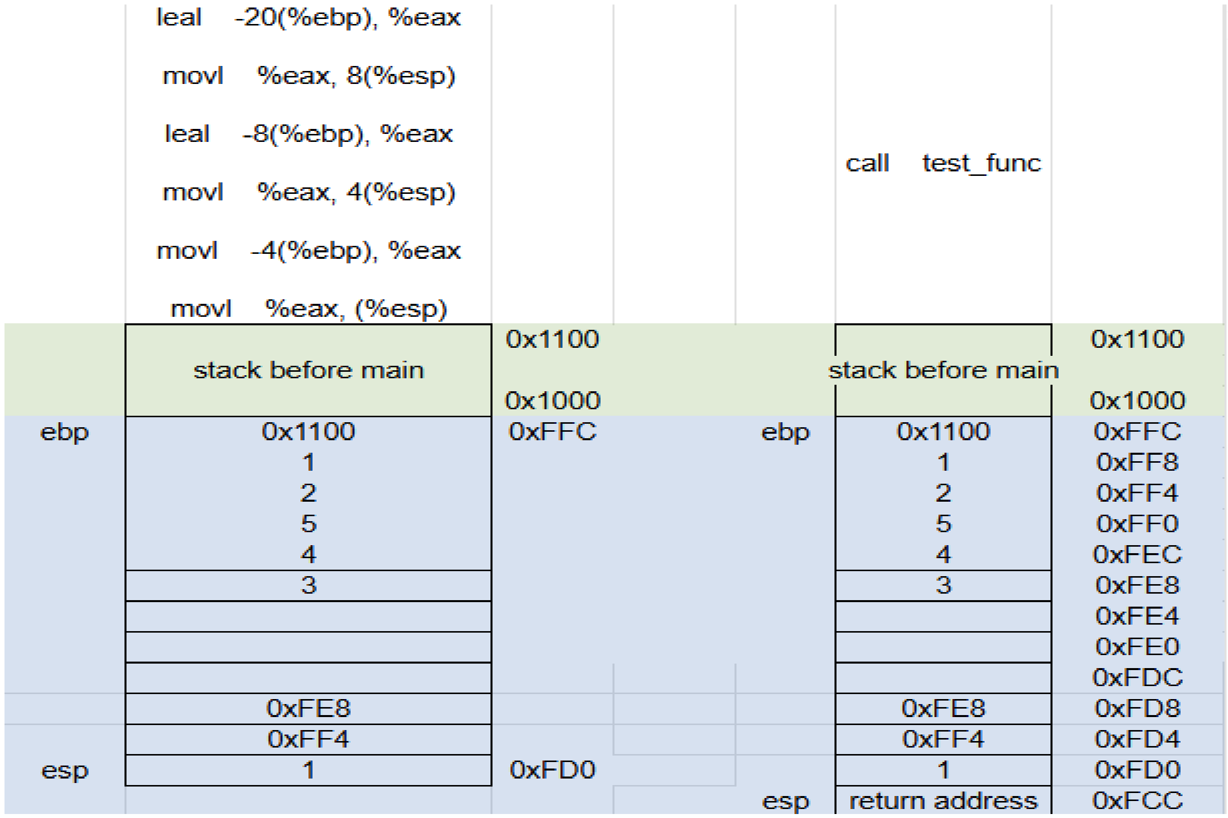


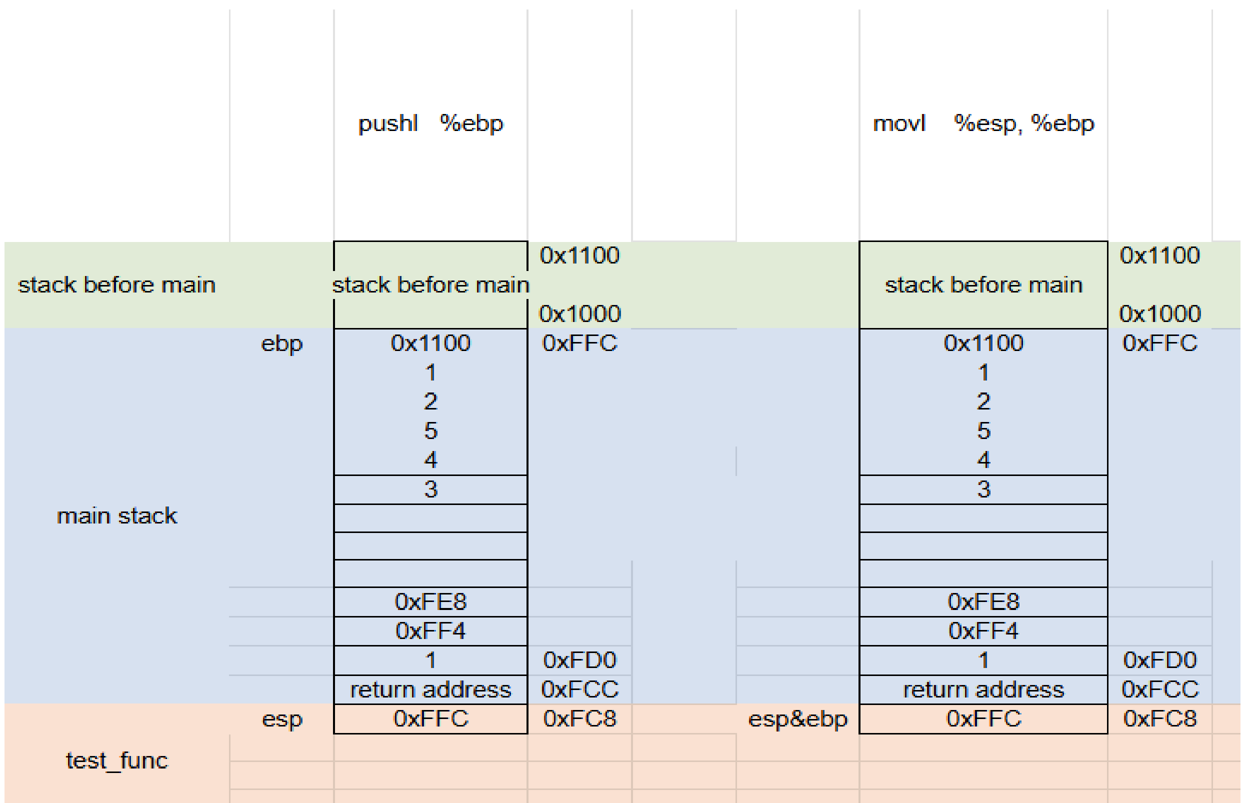

movl %esp, %ebp
stack before main
0x1100
0x1000
esp,ebp
0x1100
0xFFC


subl $44, %esp
stack before main
0x1100
0x1000
ebp
0x1100
0xFFC
esp
0xFD0


movl $1, -4(%ebp)
movl $2, -8(%ebp)
movl $3, -20(%ebp)
movl $4, -16(%ebp)
movl $5, -12(%ebp)
stack before main
0x1100
0x1000
ebp
0x1100
0xFFC
1
2
5
4
3
esp
0xFD0


leal -20(%ebp), %eax
movl %eax, 8(%esp)
leal -8(%ebp), %eax
movl %eax, 4(%esp)
movl -4(%ebp), %eax
movl %eax, (%esp)
stack before main
0x1100
0x1000
ebp
0x1100
0xFFC
1
2
5
4
3
0xFE8
0xFF4
esp
1
0xFD0


leal -20(%ebp), %eax
movl %eax, 8(%esp)
leal -8(%ebp), %eax
movl %eax, 4(%esp)
movl -4(%ebp), %eax
movl %eax, (%esp)
call test_func
stack before main
0x1100
0x1000
ebp
0x1100
1
2
5
4
3
0xFE8
0xFF4
esp
1
0xFFC
0xFD0
stack before main
0x1100
0x1000
ebp
0x1100
1
2
5
4
3
0xFE8
0xFF4
1
esp
return address
0xFFC
0xFF8
0xFF4
0xFF0
0xFEC
0xFE8
0xFE4
0xFE0
0xFDC
0xFD8
0xFD4
0xFD0
0xFCC


pushl %ebp
movl %esp, %ebp
stack before main
main stack
test_func
stack before main
0x1100
0x1000
ebp
0x1100
1
2
5
4
3
0xFE8
0xFF4
1
return address
esp
0xFFC
0xFFC
0xFD0
0xFCC
0xFC8
stack before main
0x1100
0x1000
0x1100
1
2
5
4
3
0xFE8
0xFF4
1
return address
esp&ebp
0xFFC
0xFFC
0xFD0
0xFCC
0xFC8

```
addl    $6, 8(%ebp)
```

	stack before main	0x1100 0x1000
	0x1100	0xFFC
	1	
	2	
	5	
	4	
	3	
	0xFE8	
	0xFF4	
	7	0xFD0
	return address	0xFCC
esp&ebp	0xFFC	0xFC8

```
movl    12(%ebp), %eax
movl    (%eax), %eax
leal    7(%eax), %edx
movl    12(%ebp), %eax
movl    %edx, (%eax)
```

	stack before main	0x1100 0x1000
	0x1100	0xFFC
	1	
	9	
	5	
	4	
	3	
	0xFE8	
	0xFF4	
	7	0xFD0
	return address	0xFCC
esp&ebp	0xFFC	0xFC8

```
movl    12(%ebp), %eax
movl    (%eax), %eax
leal    7(%eax), %edx
movl    12(%ebp), %eax
movl    %edx, (%eax)
```

stack before main	0x1100 0x1000
0x1100	0xFFC
1	
9	
5	
4	
3	
0xFE8	
0xFF4	
7	0xFD0
return address	0xFCC
0xFFC	0xFC8

```
movl    16(%ebp), %eax
addl    $4, %eax
movl    16(%ebp), %edx
addl    $4, %edx
movl    (%edx), %edx
addl    $8, %edx
movl    %edx, (%eax)
```

	stack before main	0x1100 0x1000
	0x1100	0xFFC
	1	
	9	
	5	
	12	
	3	
	0xFE8	
	0xFF4	
	7	0xFD0
	return address	0xFCC
esp&ebp	0xFFC	0xFC8

```
nop
popl    %ebp
ret
```

	stack before main	0x1100 0x1000
ebp	0x1100	0xFFC
	1	
	9	
	5	
	12	
	3	
	0xFE8	
	0xFF4	
esp	7	0xFD0

3. What is the problem of the following program?

```c
void get_memory(char *p)
{
    p = (char *)malloc(100);
    return;
}
int main(void)
{
    char *str = NULL;
    get_memory(str);
    strcpy(str, "Hello World!");
    printf(str);
    return 0;
}
```

   a. In get_memory, p is a local copy of the pointer str from main.
   When p is assigned the result of malloc, it only changes the local copy of p, not the original str in main.
   After get_memory returns, str in main is still NULL

```c
char *get_memory(void)
{
    char p[] = "Hello World!";
    return p;
}
int main(void)
{
    char *str = NULL;
    str = get_memory();
    printf(str);
    return 0;
}
```

   b. The problem of the following program is that the function **get_memory** returns a pointer to a local variable **p**. Local variables are stored on the stack, and their memory is automatically deallocated when the function returns. Therefore, the pointer **str** in main will point to invalid memory after **get_memory** returns.

```c
void get_memory(char **p, int num)
{
    if (p == NULL)
        return;
    *p = (char *)malloc(num);
    return;
}
int main(void)
{
    char *str = NULL;
    get_memory(&str, 100);
    strcpy(str, "Hello World!");
    printf(str);
    return 0;
}
```

c. **Incorrect Usage of printf :** printf(str); is unsafe because it can lead to format string vulnerabilities. It should use a format specifier like %s.

No Error Handling for malloc: The code does not check if malloc was successful. If malloc fails, it returns NULL, and using strcpy on a NULL pointer will cause undefined behavior.