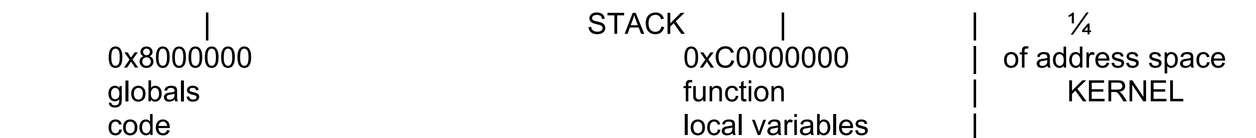
Graded assignment 0 will be handed out in section

Assignment 1 “Not that bad”
- Check your work (run it through the compiler)

Factorial Program
- Prints out ENTERING, LEAVING, and other pointers

unsigned char memory[] 2^32




```
// y.c
#include <stdio.h>

int x;

int main*() {
	printf(“%p\n”, %x);
}
```

gcc -o y y.c ; ./y
0x804a020

```
// y.c
#include <stdio.h>

int x;
int *y = &x;

int main*() {
	printf(“%p\n%p\n”, &x, y);
}
```

gcc -o y y.c ; ./y
0x804a020
0x804a020

20a00408 vs. 804a020 (LITTLE ENDIAN)

```
union a {
	unsigned char c[4];
	int i;
};
```

union a x;

x.i = 4; *//0x00000004*

Therefore:
x.c[0] = 4;
x.c[1] = 0;
x.c[2] = 0;
x.c[3] = 0;

Compiler decides the memory addresses before the program runs

- Global addresses valid as long as the programs run

STATIC DURATION variable

- Variable allocated for duration of program
- Examples:
    - Global variables
    - The code that makes up the program
- Compiler and OS decide addresses before execution

AUTOMATIC duration

- Allocated for the duration of FUNCTION
- Local variables

uintptr_t stacktop;
    An integer type large enough to hold an address

Example with factorial program:
Factorial code from before...

```
int factorial(int n) {
        int x;
        if(n==0)
                x = 1;
        else
                x = factorial(n-1) * n;
        return x;
}
```

Question: When calling factorial in main, what is happening to the variable x?

- x has automatic duration
- new x every time the function executes
- compiler must allocate space for all of a function's variables at the beginning of the function call, and deallocate said space at the function's end

We can represent this in C with a struct:

```
struct fn_args_locals {
        - each argument
        - each local variable
```

}

In the factorial example:

```
struct factorial_args_and_locals {
        int n; /* the single argument */
        int x; /* the single local variable */
}
```

Iterative rewrite of old factorial code:
We must first create a global unsigned char stack[1000000] and global variable stacktop, then within factorial_start...
1.) Subtract size of the factorial_args_and_locals struct from stacktop
2.) store a factorial_args_and_locals struct in stack at current stacktop position
3.) set argument in said struct to current argument
4.) check for base case (bottoms out at n = 0)

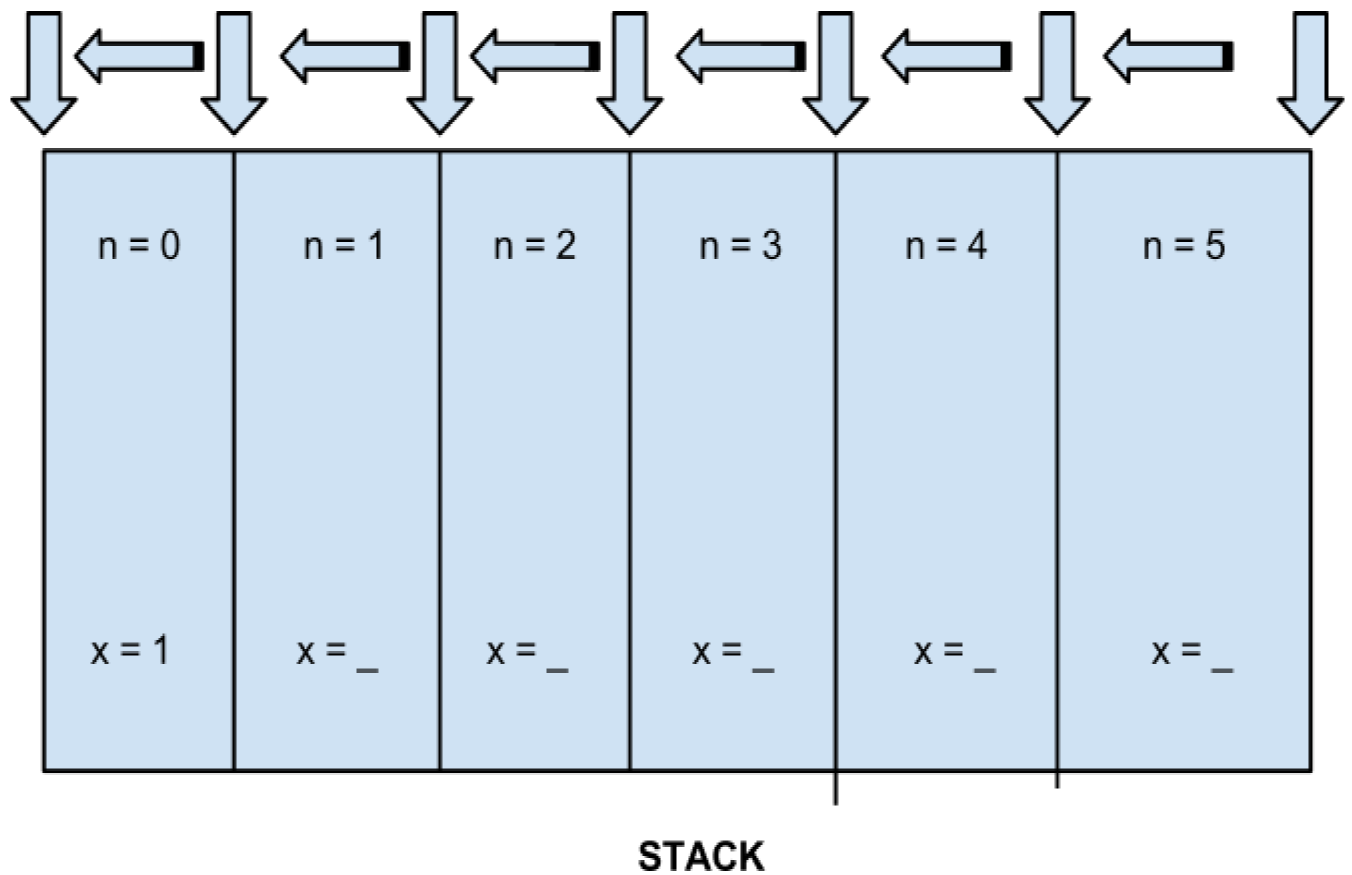
- if base case is reached, iterate back up through stack setting local variable “x” of each struct to local variable “x” from previous struct times argument “n” within struct
- else subtract 1 from current argument and goto step 1.)

Visual Representation:

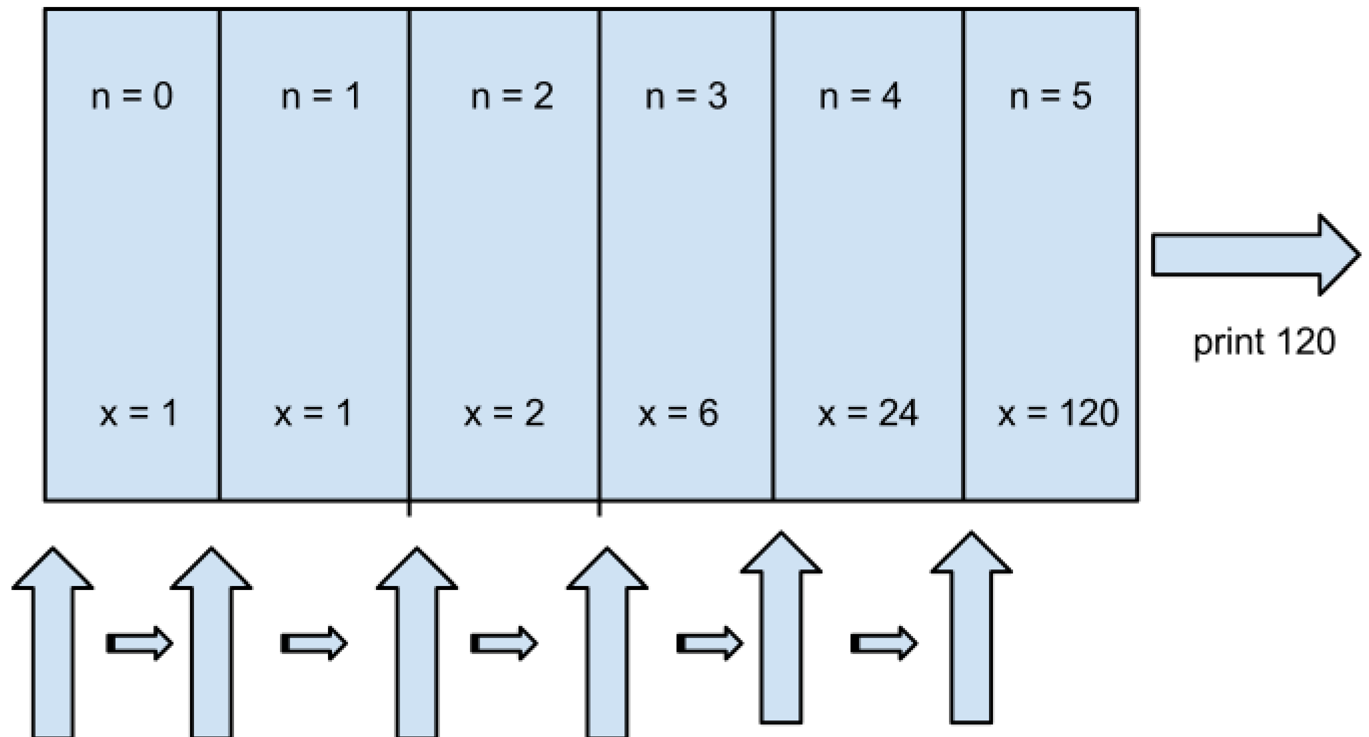
First stacktop moves down the stack in intervals of sizeof(factorial_args_and_locals);
Each struct put into the stack has

1.) int n, which is the argument
2.) int x, which is the local variable

*Note when n = 0, x is set to one due to base/special case



Now we move back up the stack by intervals of sizeof(factorial_args_and_locals) setting x = previous x * n for each struct.



Final Code:

```
struct factorial_args_and_locals {
	int n;
	int x;
};
unsigned char stack[1000000];

int main(int argc, char **argv) {
	int n = 2;
	if(argc >= 2)
		n = strtol(argv[1], 0, 0);
	fprint(stderr, “%d!=”, n);
	uintptr_t stacktop = &stack[1000000];
	int x;
	int first_n;

	struct factorial_args_and_locals *l;
factorial_start:
	stacktop -= sizeof(struct factorial_args_and_locals);
	l = (struct factorial_args_and_locals *) stacktop;
	l->n = n;
	if (l->n == 0)
		l->n = 1;
	else {
		n = l->n - 1;
		goto factorial_start;
	factorial_returns_here:
```

```
        l->x = x * l_>n;
    }
    if (l->n != first_n) {
        x = l->x;
        stacktop += sizeof(struct factorial_args_and_locals);
        l = (struct factorial_args_and_locals *) staktop;
        goto factorial_returns_here;
    }

    printf("%d! == %d\n", n, x);
}
```

Last 3rd of lecture:

Why do we call it a stack?

Use stack of paper as an analogy. A stack of paper is orderly, and it's easier to access papers on top of the stack (e.g. remove and add papers to top of stack). This similarly applies to the stacks we reference in class.

Heap:

In class demonstration with program in_dictionary

- What type of duration does this program use?
    - Static storage duration for its code.
    - Automatic storage duration for its local variables and arguments.
- Program time to confirm word is in dictionary is proportional to the size of the dictionary. (Finding 2 words took 0.03 seconds)
- Why is this program running so slow?
    - It reads data in from dictionary file each time (for each word).
        - Why? Because program put dictionary inside function, which means it has automatic storage duration. So each time function returns, everything gets "lost."
    - To solve this, we want to create a global variable.
        - e.g. static char dictionary[1000000]
    - Need 3rd class of memory storage that isn't pre-allocated by compiler but can outlast the duration of a function. This illustrates the need for DYNAMIC STORAGE DURATION.

Dynamic storage duration:

- Variables outlast current function
- Memory used to hold variables is not controlled by the compiler, it's controlled by the user
- How do we get access to dynamic storage duration variables?
    - Malloc - allocates sz bytes of space w/ dynamic storage duration. Guaranteed to not overlap with anything else that's active in the program.
    - Free - to get rid of an object with dynamic storage duration. Deallocates object at location pointer so it can be used later
- What's an object of dynamic storage duration for which no pointers exist? Garbage.
- Where do objects with dynamic storage duration go? The heap.

Look at some memory allocation related errors:

- Test 28.c
    - Where's the error?

  - int *array = (int *) malloc(10);
  - This allocated 10 bytes of memory, but following lines of code are initializing all 10 elements of array to 0, but ints have 4 bytes, so we haven't allocated enough memory

End note: How would you write the following debugging tool?
    Valgrind - gives feedback as to memory allocation errors