Calling convention. This is how the C compiler matches functions to instructions. Why is this called a convention, and not a rule or law? It's because there can actually be compilers that don't use the stack (and some compilers don't!) or that support weird functions (functions that return more than once, or continue a function from a savepoint...). Conventions like this allow multiple compilers to work together.

Examples. `%eax` is return value. First argument is at `4(%esp)` when function begins. Return address is at `(%esp)`.

Inside a function. `%esp` = TOP of stack. May change as function executes. `%ebp` = boundary between parameters and local variables. Does not change as function executes.

Callee-saved registers. Let's say that `main` calls `f(int a, int b)`. The caller needs to have some guarantees about some registers that will stay the same. These are the registers that the callee needs to save if the callee wants to use them. Examples: `%ebp`, `%esp`, `%ebx`, `%esi`, and `%edi`. Caller-saved registers are `%eax` (which will be used for the return value), `%ecx`, and `%edx`.

What does `pushl` do? `pushl x` ≡ `subl $4, %esp; movl x, (%esp)` and `popl x` reverses this.

How are local variables referred to? Addresses like `-4(%ebp)`, etc.

Creating a new stack frame. Next, the example program runs `subl $8, %esp`. This creates a new stack frame. But there are no arguments being passed, so why? *The calling convention requires that stack frames are 16-byte aligned!*

What does `leave` do? It's shorthand for `movl %ebp, %esp; popl %ebp`. Sometimes the compiler in its smartness realizes it doesn't use `%ebp`, so it omits this and the `pushl` at the top.

Order of arguments. Arguments are pushed onto the stack in reverse order: the last argument has the highest address, the second-to-last has the next highest, and so on.

`leal`? This instruction means "load effective address" – takes something that looks like a dereference, but doesn't do the dereference, storing the address somewhere else. It's basically an arithmetic instruction.

Loops. Compilers do a lot of optimizations. For example, `while` loops (loops that run 0 or more times) are often compiled into a branch and a `do ... while` loop (that runs 1 or more times). `while` loops with conditions containing `&&` are often compiled into loops with 2 exits.

Infinite loop attack. `.label: jmp .label`. Why doesn't this freeze up the processor? We'll see next time.