Function Implementation (x86-specific)

- double words are pushed and popped
- dedicated register %esp contains address of item currently at top of stack (TOS)
pushl *

does %esp ← %esp − 4
movl *, (%esp)

popl *

does movl (%esp), *
%esp ← %esp + 4

THE STACK
The Stack

push $6
push $-1
push $3

THE STACK
Layout of segments in memory

- code
- data
- heap
- stack

addr 0
What we need to know how to do. . .
(what the compiler must be able to implement)
1. call
2. return
3. AR and local variables
4. return value
5. parameters
1. call

- remember the return address
- go to fcn

This is such a common operation that the x86 architecture supports it with a single instruction:

```
call fcn
```

does the equivalent of

```
push %eip
jmp fcn
```
2. return

use the return address pushed onto the stack

ret

does the equivalent of

popl  %eip
void c() {
    // c
}

void b() {
    c();
}

void a() {
    b();
}

main() {
    a();
}

sample code to show only call and return

void c() {
    // c
}

void b() {
    // c
    // call c
    L3:   // ret
}

void a() {
    // call b
    // a
    L2:   // ret
}

main() {
    // call a
    // main
    L1:   // ret

addr 0
%esp
3. incorporate AR

For example, assume we need AR space for 3 ints. gcc on x86 allocates AR space in multiples of 16 bytes.
prologue code
pushl %ebp
movl %esp, %ebp
subl $16, %esp

Before fcn starts, but after the call instruction

After fcn prologue

%esp
prev %ebp
ra
%ebp
addr 0
epilogue code

leave

does movl %ebp, %esp

popl %ebp

ret
does popl %eip
Put local variables into AR:

```c
void b() {
    int x, y, z;
    x = 1;
    y = 2;
    z = 3;
    c();
}
```

```
b:  pushl %ebp  prologue
    movl %esp, %ebp
    subl $16, %esp
    movl $1, -12(%ebp)
    movl $2, -8(%ebp)
    movl $3, -4(%ebp)
    call c
    leave  epilogue
    ret
```
4. return value
On x86, return value goes in %eax (by convention)

```c
int b() {
    c();
    return 4;
}
```

```assembly
b:
    call c
    movl $4, %eax
    leave
    ret
```
5. parameters

No room in registers on the x86, so parameters go onto the stack.
Caller allocates space and places copies (for call by value). Child retrieves and uses copies.

```
main () {
    a( 1, 2, 3 );
}
main:    pushl %ebp
           movl %esp, %ebp
           subl $12, %esp
           movl $1, (%esp)
           movl $2, 4(%esp)
           movl $3, 8(%esp)
           call a
           leave
           ret
```
main's outgoing parameters

 addr 0

 p1
 p2
 p3

 ⬤ ⬤ ⬤ %ebp
int b(int a1, int a2) {
    return (a1 + a2);
}

void a(int p1, int p2, int p3) {
    b(4, 5);
}

```c
int b(int a1, int a2) {
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    movl 8(%ebp), %edx
    add %edx, %eax
    popl %ebp
    ret
}

void a(int p1, int p2, int p3) {
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $4, (%esp)
    movl $5, 4(%esp)
    call b
    leave
    ret
```
by convention, and to make the compiler's job easy, the first parameter will *always* be at 8(%ebp)
current AR

P1 in
P2 in
P3 in

P1 out
P2 out
P3 out

prev %ebp

ra

local variables
callee-saved regs

P1 incoming is always at 8(%ebp)

(addr 0)
%ebp
%esp

(parameter build space)