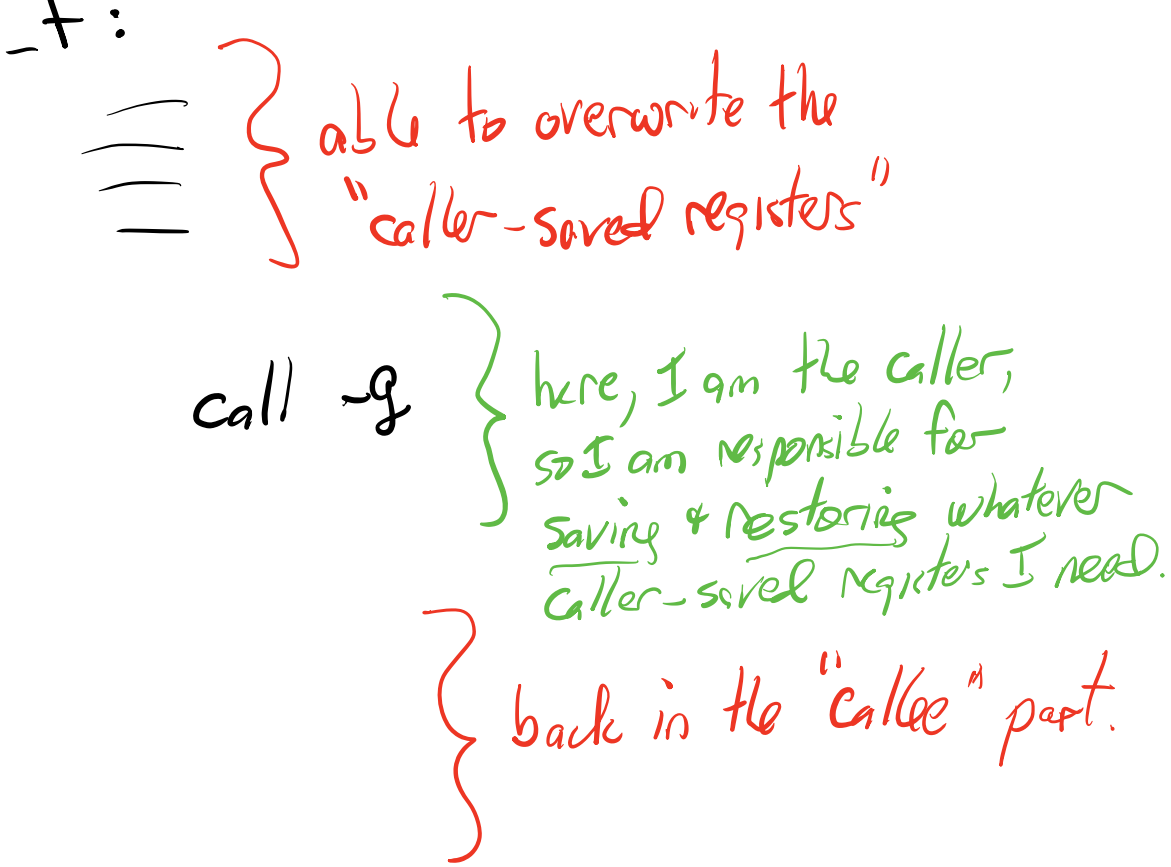
Caller-saved, callee-saved registers

- when you are writing a function in assembly, that function will be called by another function, so your function is the "callee".



How do I save a caller-saved register?

1. Move its contents into a callee-saved register
2. Push its contents on the stack.

caller saved

```
mov $24, %rcx
push %rcx   # save on stack
call -g
pop %rcx    # restore from stack
add $12, %rcx
```

Saving multiple registers?

- pop in reverse order of push.

```
push %rcx
push %rdx
call -g
pop %rdx
pop %rcx
```

Important: push and pop only work on 64-bit values.

A callee-saved register must be saved before you use it and restored after you're done with it.

- caller expects these registers to be unchanged.

```
f:
    push %rbx   # save %rbx
    mov $100, %ebx   # callee
                     # saved
    ...
    # done using %ebx
    pop %rbx    # restoring it.
```

Typically, the callee-saved registers you are going to use are saved at the beginning of the function (after the "push %rbp" and "mov %rsp, %rbp").

The caller-saved registers that you need to save are typically saved before a function call and restored after the call returns.

Pointers in assembly

C code:


```
int foo(int *p)
{
  *p = *p * 2;
}

int main()
{ int x = 20;
  foo(&x);
  printf("%d\n", x); // 40
```

In assembly:

```
_foo:   push %rbp
        mov  %rsp, %rbp
        # p is in %rdi
        imul $2, (%rdi)
             # *p = *p * 2
        pop  %rbp
        ret
```

Passing parameters, revisted

Mac/Linux: First 6 parameters are passed in registers, rest passed on the stack in reverse order.

Windows: First 4 in registers, rest on stack in reverse order.

See the calling convention document on Brightspace.

C code:

```
int bar(int a, int b, int c,
        int d, int e, int f,
        int g, int h, int i,
        int j);

int main()
{ int x;
  x=bar(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
}
```

10 params so 4 on stack (for Mac and Linux)

How does bar access the last 4 parameters on the stack?

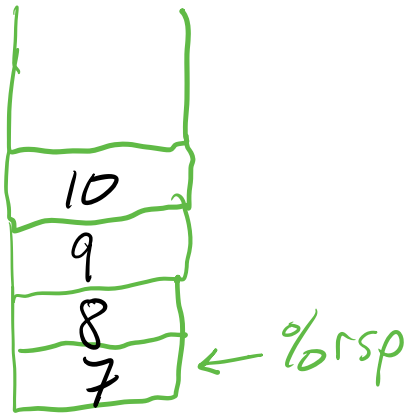
- main() pushes the last 4 parameters in reverse order:

- main:

```
push $10
push $9
push $8
push $7
```

all 64-bits

What does the stack look like now?

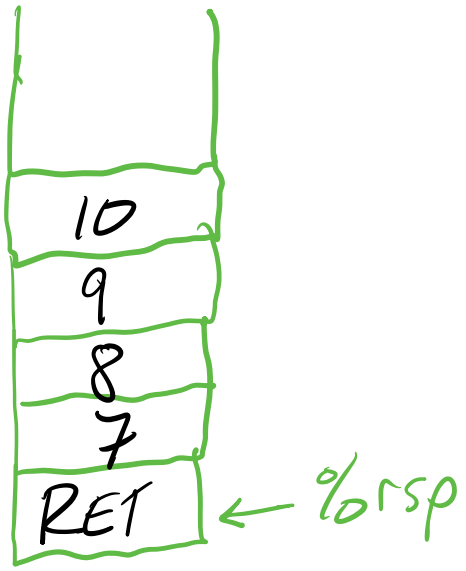


Then main() contains a call instruction:

call _bar

- pushes the return address onto the stack, jumps to bar.

So, the stack looks like:

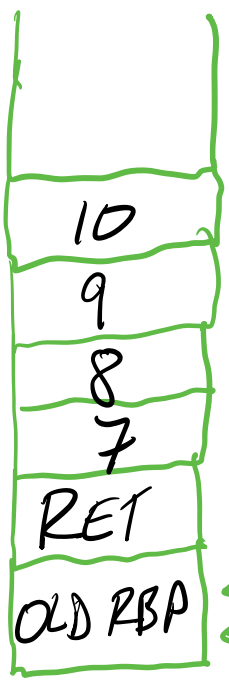


Now the code for bar() starts executing:

%rbp is callee-saved

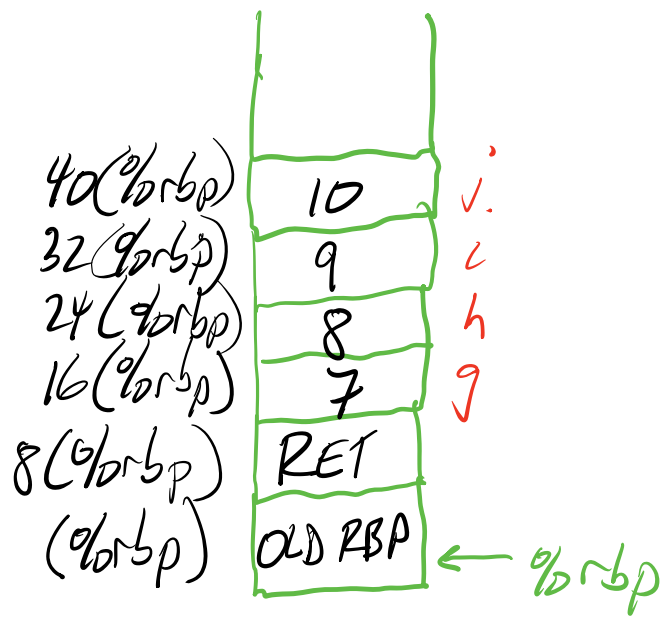
```
_bar:
    push %rbp
    mov %rsp, %rbp
```

Now the stack looks like:



%rsp, %rbp point to the same place.

We access parameters on the stack using offsets from %rbp

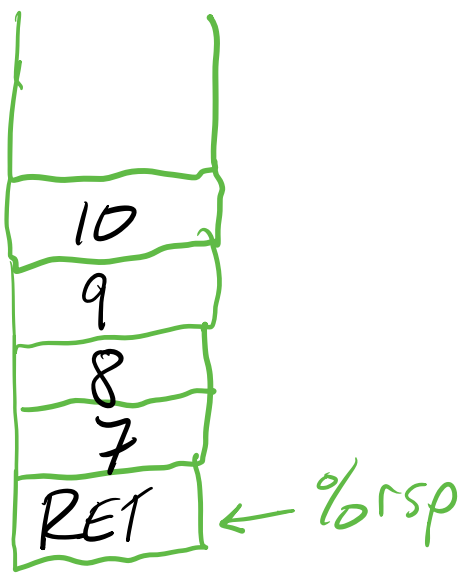



```
# suppose bar adds those last 4
# parameters together.
# Each parameter is actually
#  4 bytes, having been
# declared as "int".
```

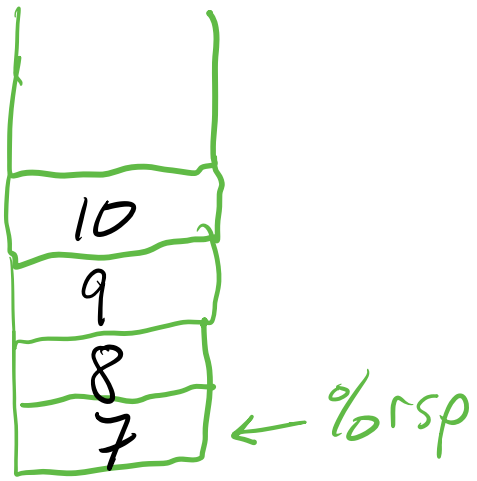
```
movl   16(%rbp), %eax   # %eax = g
addl   24(%rbp), %eax   # %eax += h
addl   32(%rbp), %eax   # %eax += i
addl   40(%rbp), %eax   # %eax += j

# we're done in bar, need to
# restore the old %ebp value and
# return back to main().

        pop %rbp
```

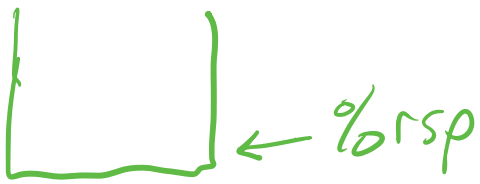


ret # return instruction



Now we're back in main()!
- it needs to remove those 4 parameters from the stack.
- how? It could do 4 pops, but its more efficient just to add some constant to %rsp.

addq $32, %rsp # adding 32 to %rsp



The stack is now in the same state it was before main() started the call to bar.