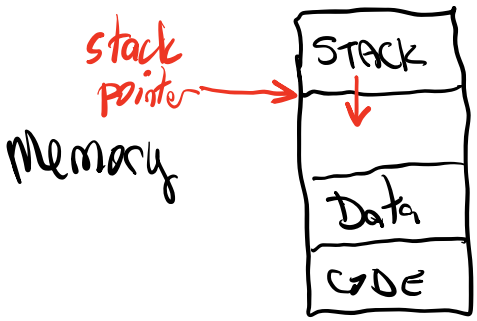
Functions use the stack



The stack grows in a downward direction.

- push operation decrements the stack pointer (and writes data into the stack)
- pop operation copies the data from the stack to somewhere else and increments the stack pointer

The stack pointer is the %rsp register.

push %rax // copies %rax into the
// stack and decrements
// %rsp.

pop %rcx // copies what was at the
// top of the stack into
// %rcx and increments
// %rsp.

What happens when a function is called?

- Suppose f calls g

C:

```
void f(...)
{ ...
  g(..);
  ...
}
```

equivalent

Assembly:

```
_f:
    .....
    call _g
    ....
```

The "call" instruction:

1. pushes the return address on the stack.
2. Jumps to the specified label.

The return address is the address of the instruction immediately after the call instruction.

```
call -q
→ addq %eax, %ecx
```

The return address pushed by the call, in this case, is the address of the addq instruction.

What does g look like?

C:

```
void g(...)
{
    ⋮
    return;
}
```

Assembly:

```
-g:  ...
     ...
     ...
     ret    //return instruction
```

The ret instruction pops the return address off the stack and then jumps to the return address.

How do parameters get passed when a function is called?

- Determined by a "calling convention"
  - an agreement among the vendors of compilers and operating systems.
  - different for Unix (macOS & Linux) and Windows (Microsoft)

Unix: First 6 parameters are passed in registers:

%rdi, %rsi, %rdx, %rcx, %r8, %r9

- the rest are pushed onto the stack in reverse order (right to left)

Windows: First 4 parameters are passed in registers:

%rcx, %rdx, %r8, %r9

- the rest are pushed onto the stack in reverse order.

- If a parameter is 32 bits, use the 32-bit half of the above registers (%edi, %esi, ...)

The return value is put in %rax (or %eax for 32-bit value).

When I'm writing a function, which registers can I overwrite?

- don't want to destroy data in a register that another function is using
  - there is only one of each register

The calling convention also tells us which registers we can overwrite.

Some terminology:

```
void f(...)
{
  ...
  g(...);
}
```

f is the "caller"

g is the "callee".

```
_f:
  ...
  call _g
  ...
```

"caller"

"callee"

```
_h:
  ...
  call _f
  ...
```

"caller"

"callee".

Calling convention indicates:

1. "Caller Saved" registers
   - the callee can overwrite these registers.
2. "Callee Saved" registers
   - when the callee returns, these registers must contain the same value that they had before the call.

What is the impact of having a caller saved register?

- the caller has to make sure that either:
  - the caller doesn't need the value in the register, or
  - the caller <u>saves</u> the value of the register before the call and <u>restores</u> the register after the call.

```
_f:
    ...
    mov $25, %eax
    call _g
```

caller saved

does %eax still contain 25?

No! (can't be sure)