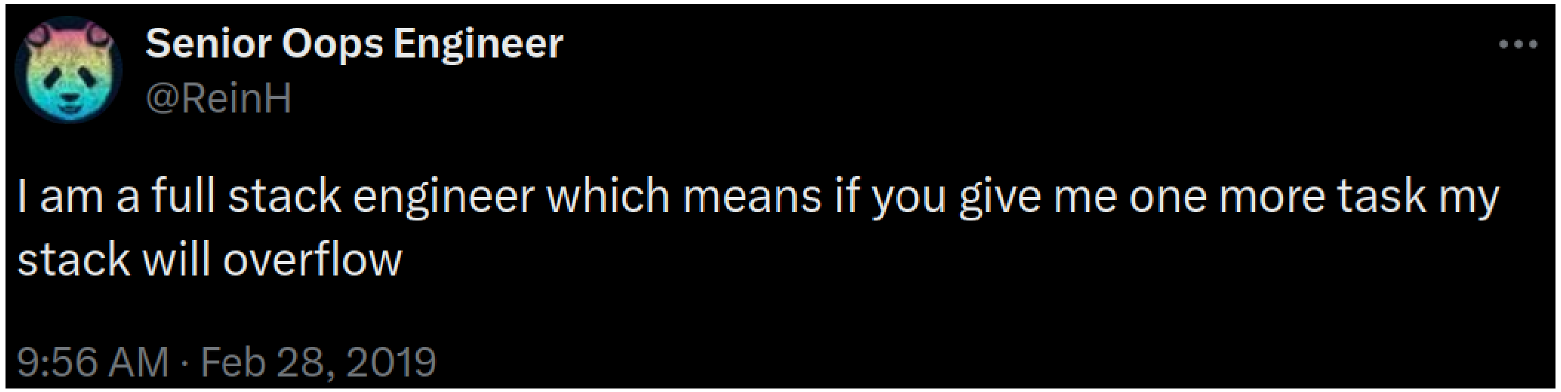
Stack & Procedures I



Senior Oops Engineer
@ReinH

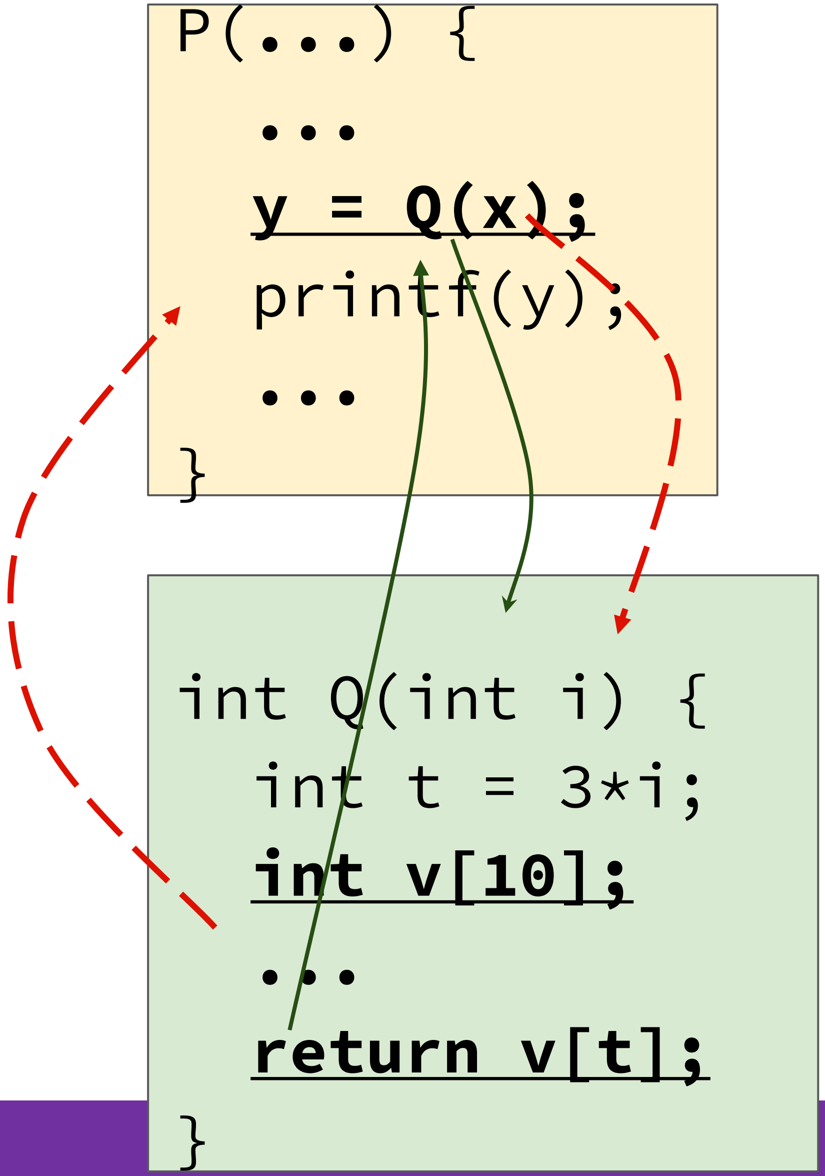
I am a full stack engineer which means if you give me one more task my stack will overflow

9:56 AM · Feb 28, 2019

Mechanisms Required for *Procedures*

1. Passing control
   - To beginning of procedure code
   - Back to return point
2. Passing data
   - Procedure arguments
3. Memory management
   - Allocate local variables during procedure execution
   - Deallocate on return

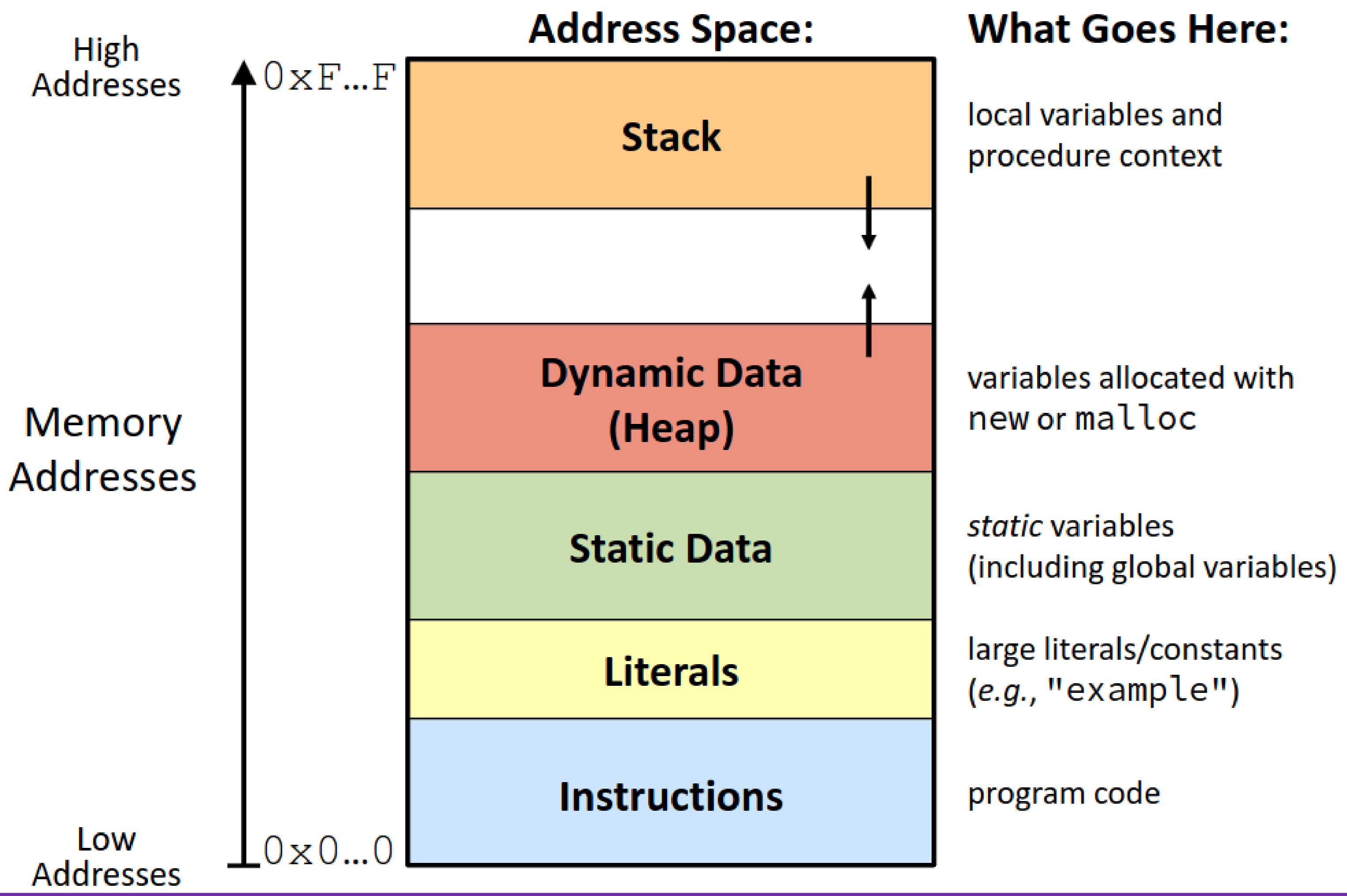
- All implemented with machine instructions!



```
P(...) {
  •••
  y = Q(x);
  printf(y);
  •••
}
```

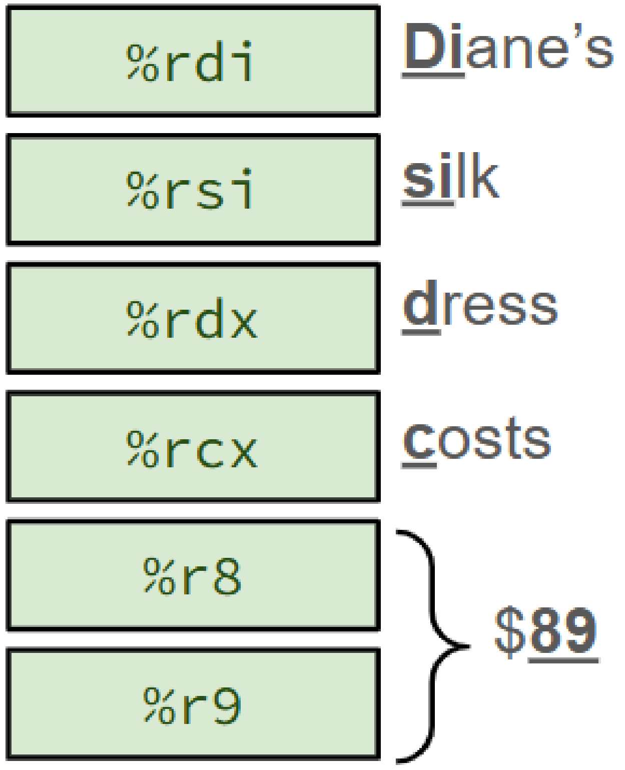
```
int Q(int i) {
  int t = 3*i;
  int v[10];
  •••
  return v[t];
}
```

Simplified Memory Layout


High Addresses
0xF…F
Memory Addresses
Low Addresses
0x0…0
Address Space:
Stack
Dynamic Data (Heap)
Static Data
Literals
Instructions
What Goes Here:
local variables and procedure context
variables allocated with new or malloc
static variables (including global variables)
large literals/constants (*e.g.*, "example")
program code

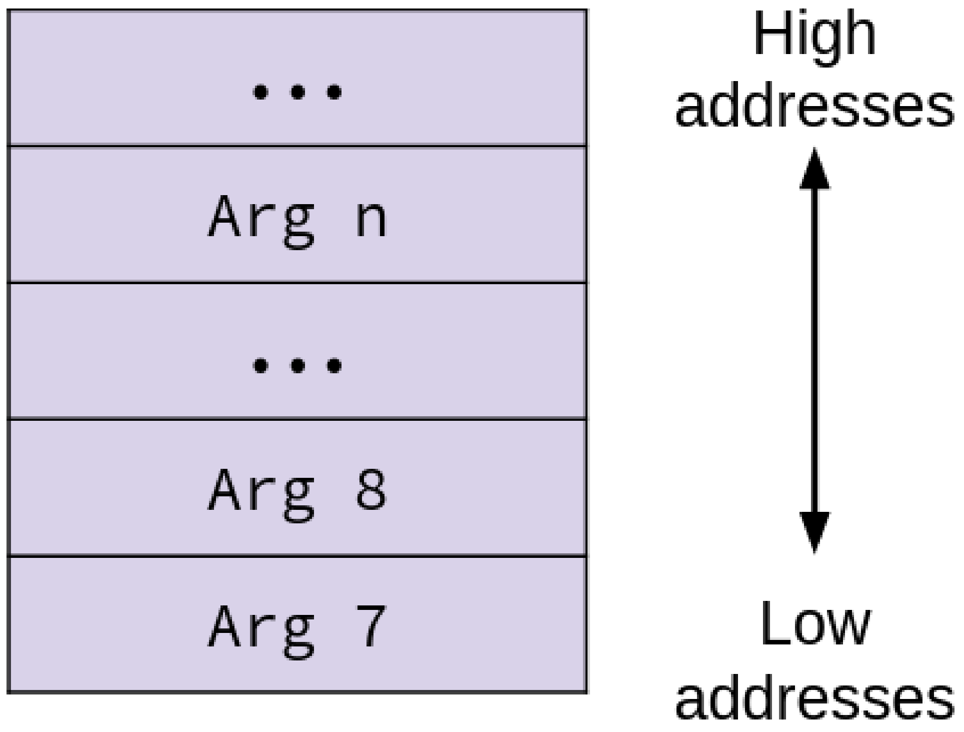
Passing Arguments

First 6 args: Registers (NOT in memory)



Extra args: Stack (Memory)

- Only allocate when needed



Return Values

By convention, stored in `%rax`

1. **Caller** must make sure to save old contents of %rax before calling a function
   - Clears out space so callee can put the return value there
   - Part of the register saving conventions
2. **Callee** places return value into %rax before return
   - Any type <= 8B (pointer, integer, etc.)
   - For larger values (ex: array), returns a *pointer* to the data
3. Upon return, **caller** finds the value in %rax

Local Data Storage

- Compiler will usually try to store local variables in **registers**
  - Faster to access than memory
- Otherwise, local data goes on the **stack**
  - Common reasons why the compiler may choose to put data in the stack:
    - No registers available
    - Data is too large (ex: arrays)
    - Variable needs to have an address (ex: C code uses the & operator)
    - Other reasons (sometimes compilers do things we don’t understand!)
- Programmer can’t accurately predict where their data will be stored ¯_(ツ)_/¯

Stack-Based Languages

- *e.g.*, C, Java, most modern languages
- Support recursion
  - Code must be *re-entrant*
    - Allow multiple simultaneous instances of the same procedure
- Stack allocated in **frames**
  - State for a single instance of a procedure
- Stack “discipline”
  - Maintained by the compiler
  - State for a given procedure is only needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

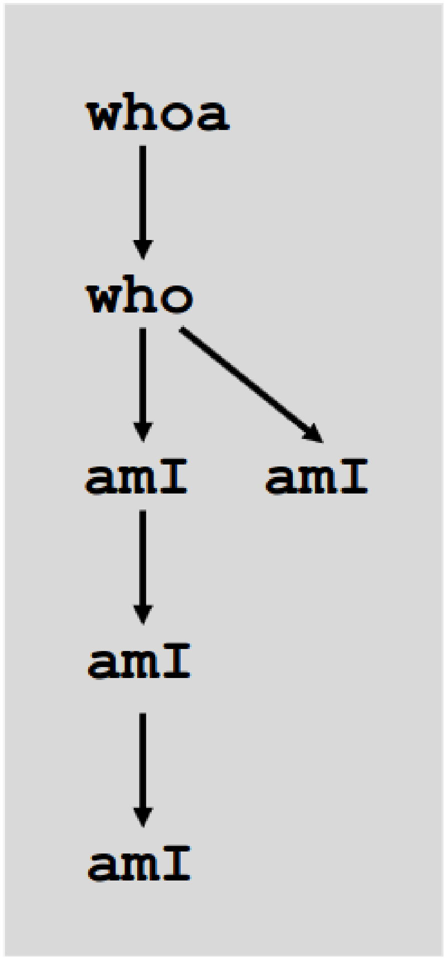
Call Chain Example

```
whoa(...)
{
   •
   •

who(...);
   •
   •
}
```

```
who(...) {
   •

amI(...);
   •

amI(...);
   •
}
```

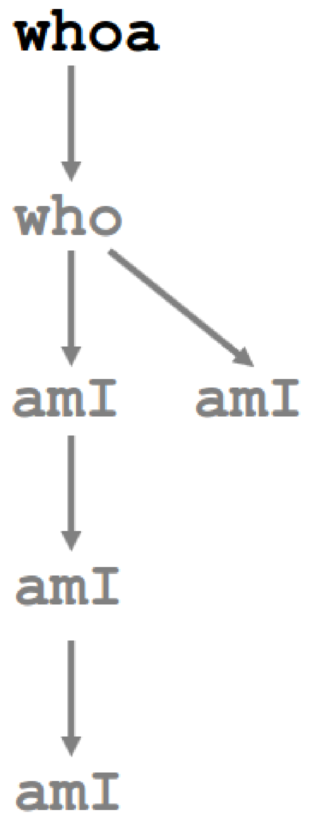
```
amI(...) {
   •
   •
  if(...)

amI(...);
   •
   •
}
```

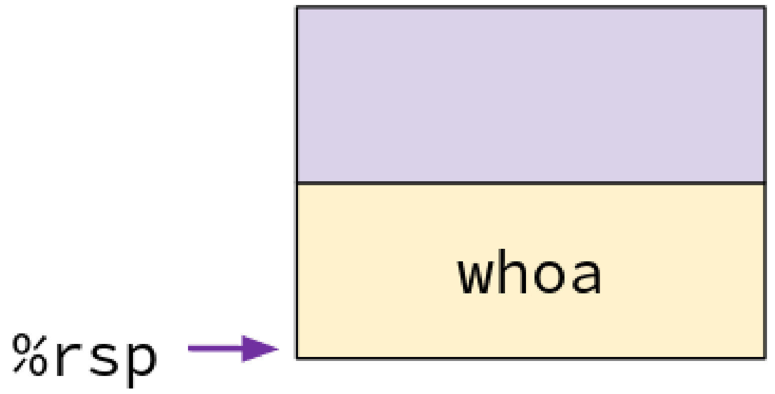
Example
Call Chain


$$\begin{tikzcd}
\texttt{whoa} \arrow[d] & \\
\texttt{who} \arrow[d] \arrow[dr] & \\
\texttt{amI} \arrow[d] & \texttt{amI} \\
\texttt{amI} \arrow[d] & \\
\texttt{amI} &
\end{tikzcd}$$

1. Call to `whoa`

```
whoa(...)
{
   •
   •

who(...);
   •
   •
}
```

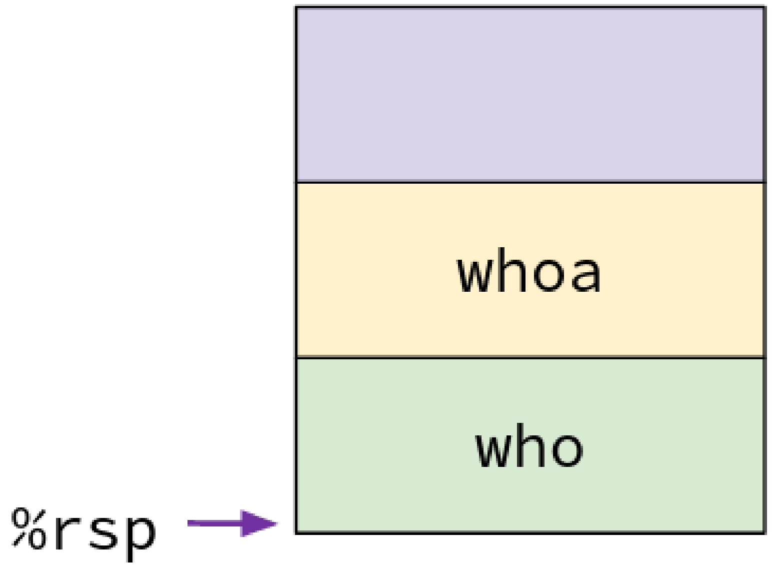




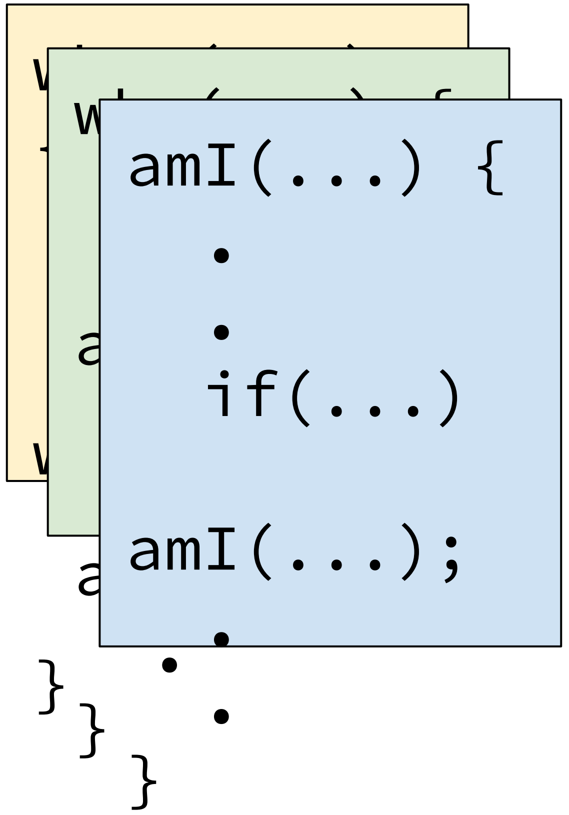
2. Call to who

```
who(...) {
  •
amI(...);
  •
amI(...);
  •
}
```

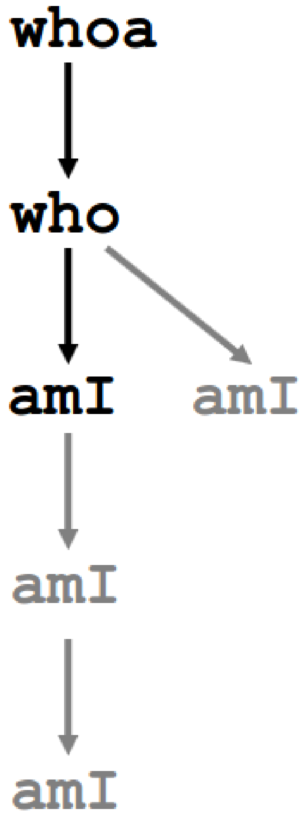
$$\begin{tikzcd}
\text{whoa} \arrow[d] & \\
\text{who} \arrow[d] \arrow[dr] & \\
\text{amI} \arrow[d] & \text{amI} \\
\text{amI} \arrow[d] & \\
\text{amI} &
\end{tikzcd}$$

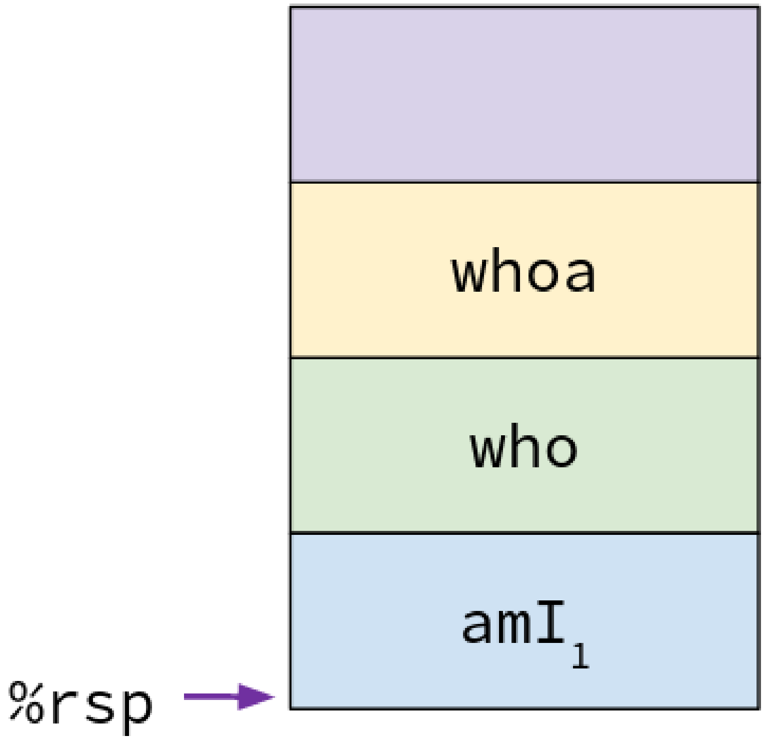


3. Call to ami

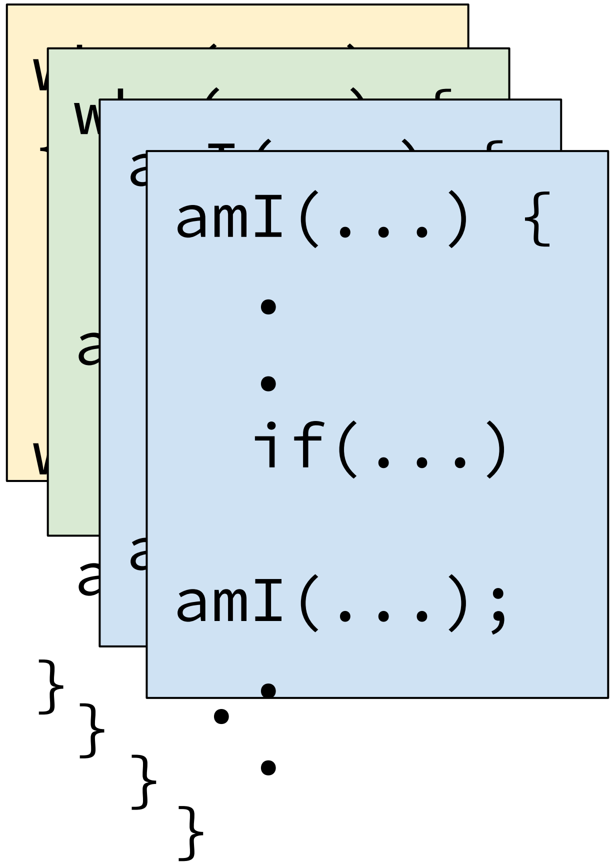


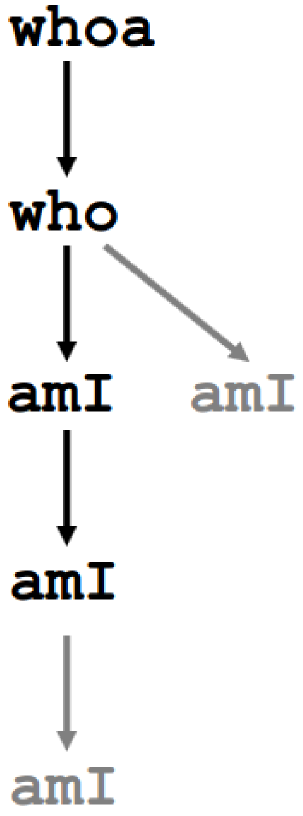
```
amI(...) {
   .
   .
   if(...)

amI(...);
   .
   .
}
```

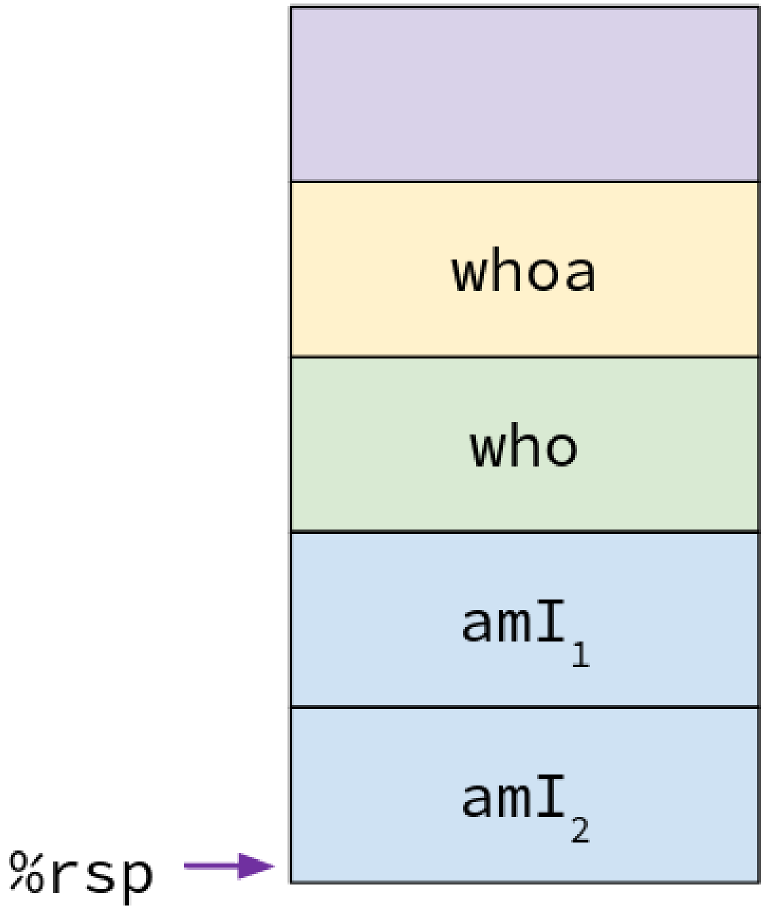




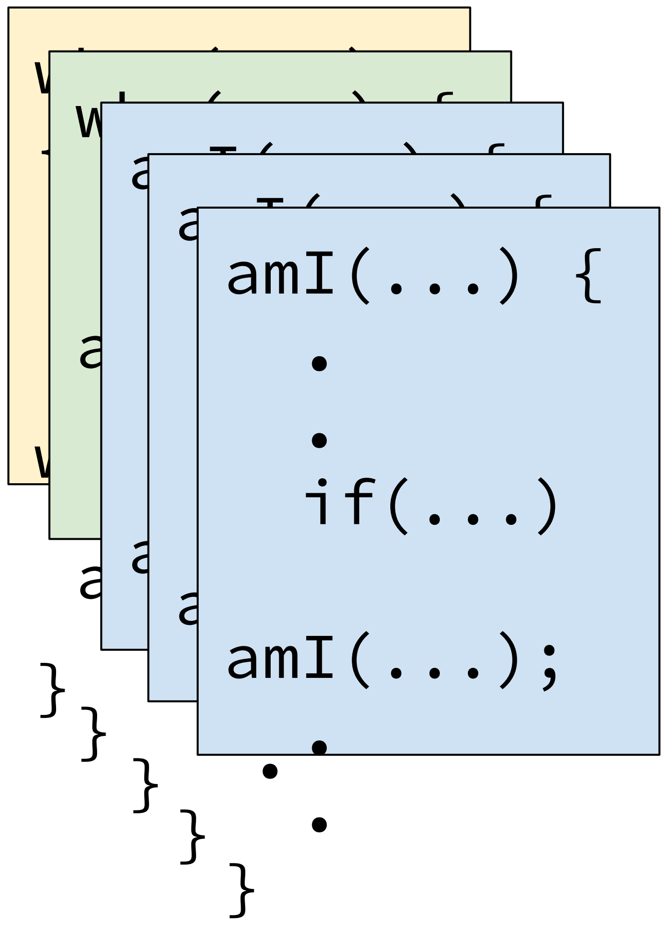
4. Recursive Call to `ami`


```
amI(...) {
   •
   •
   if(...)

amI(...);
   •
   •
   •
}
```

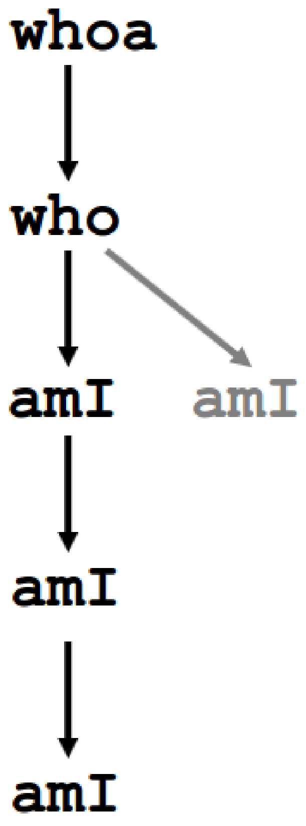


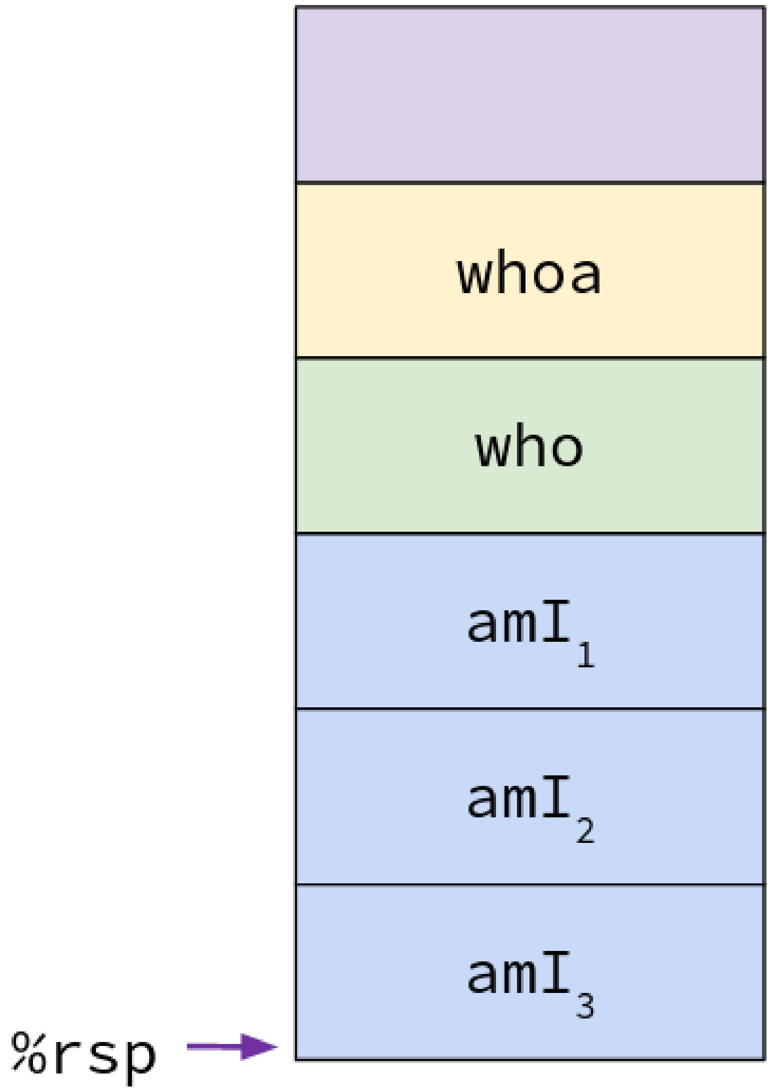


5. (another) Recursive Call to `amI`

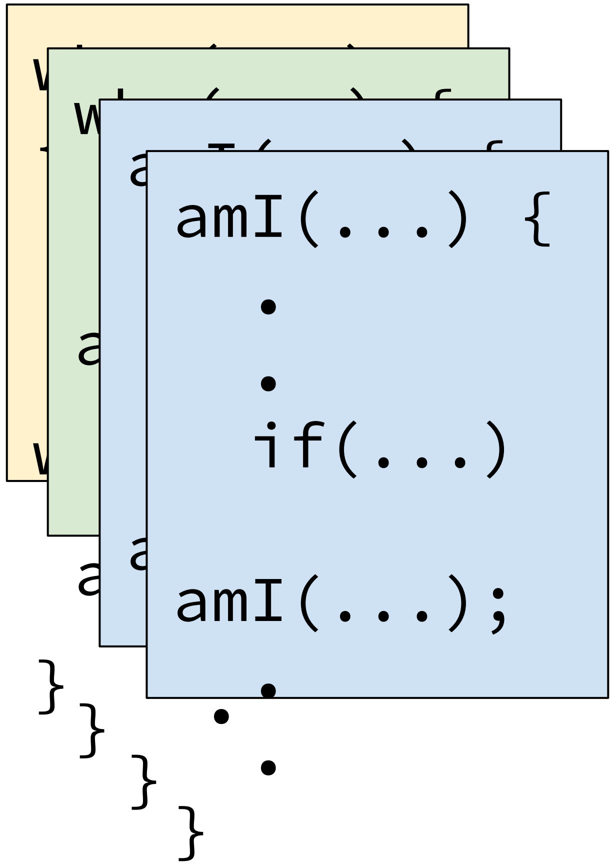


```
amI(...) {
   •
   •
   if(...)

amI(...);
   •
   •
}
```

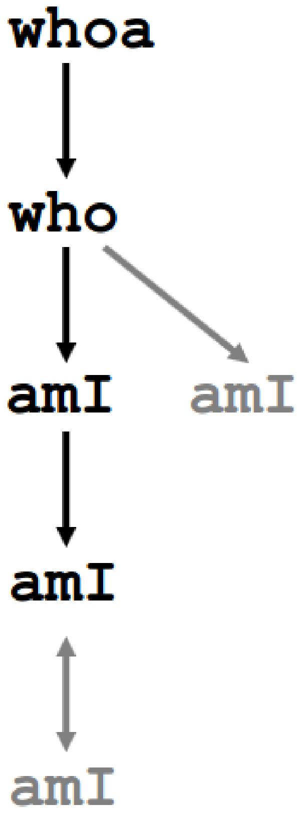


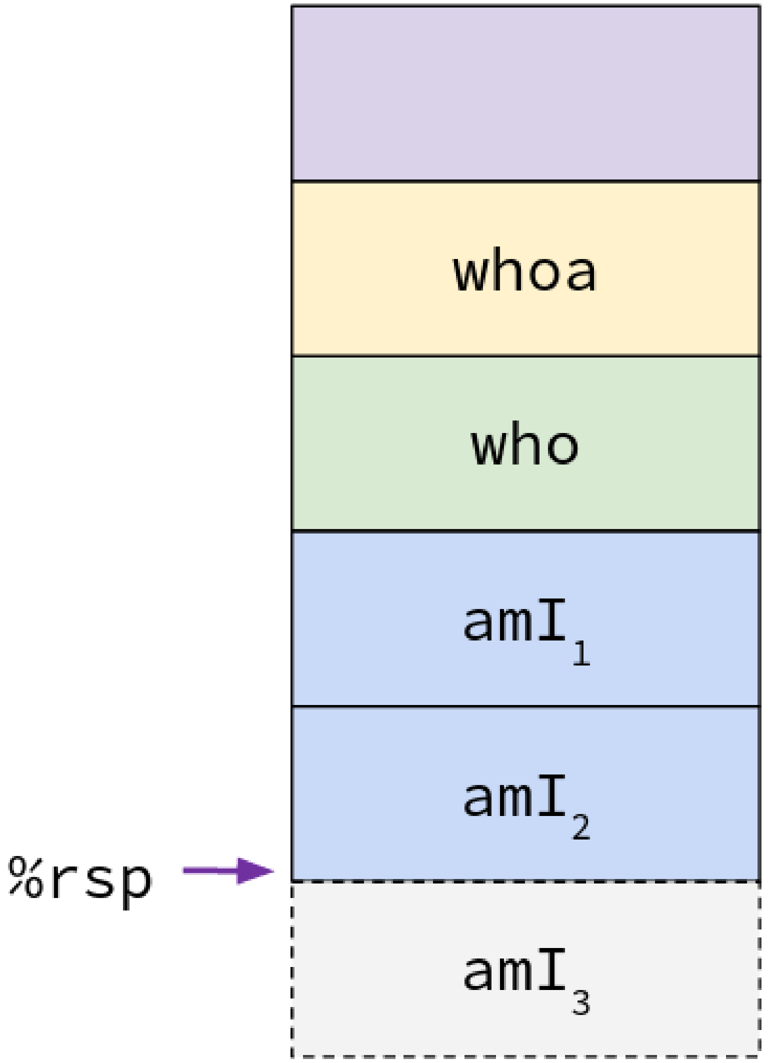


6. Return from (another) Recursive Call to **`amI`**

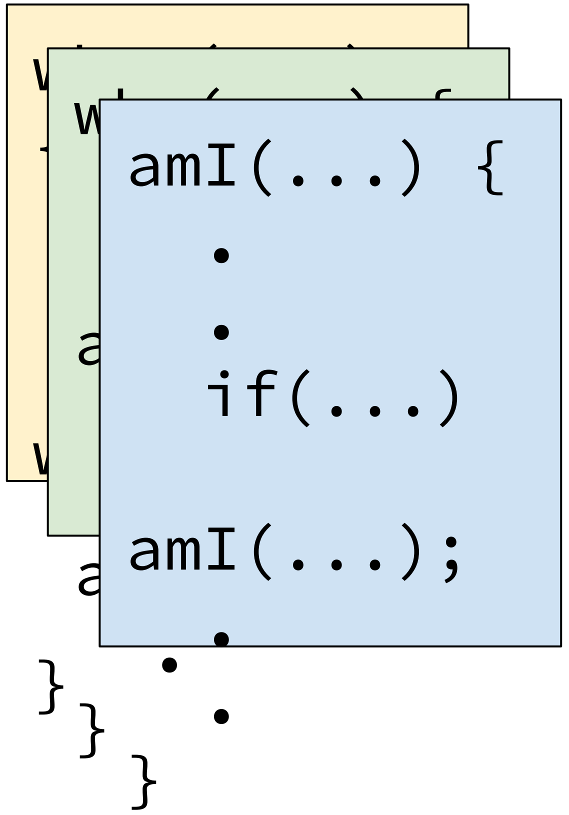


```
amI(...) {
  •
  •
  if(...)

amI(...);
  •
  •
}
```

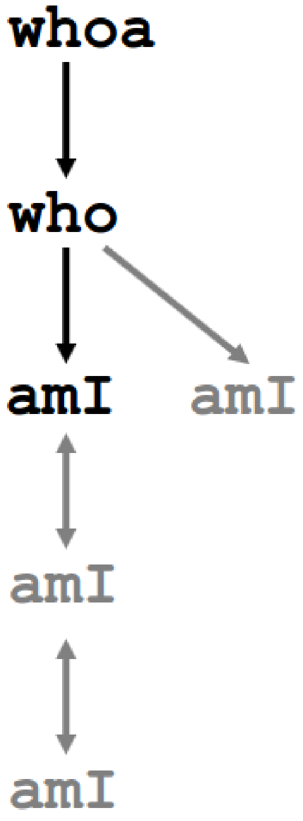


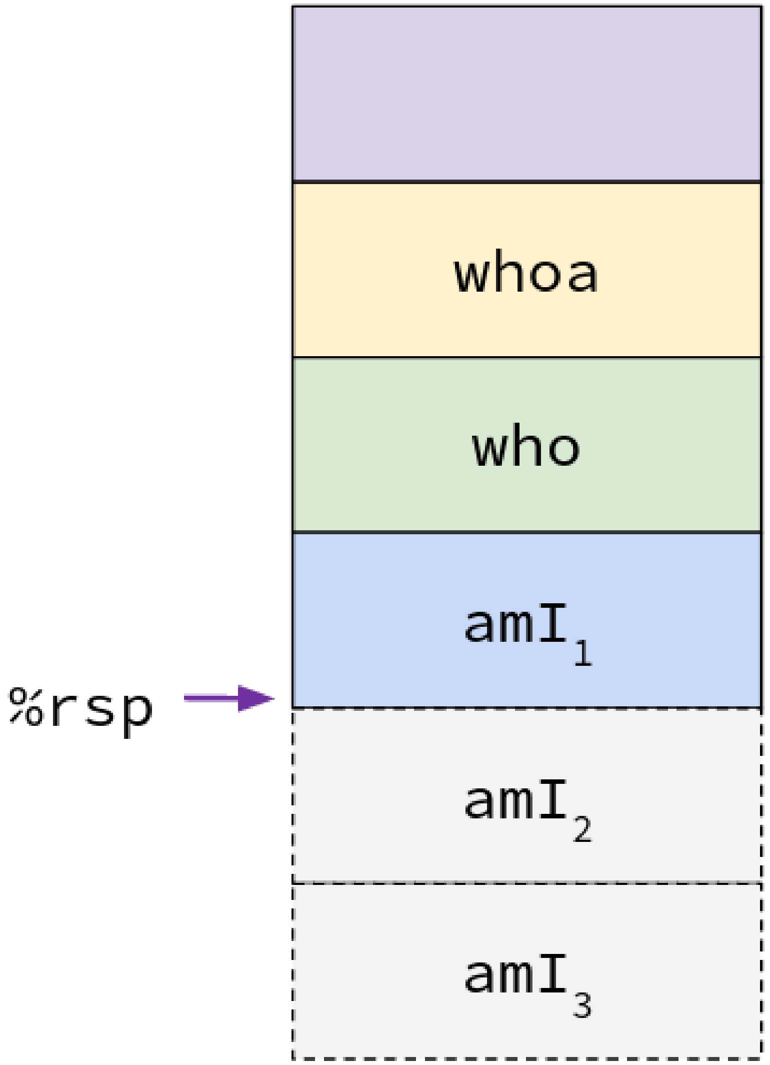


7. Return from Recursive Call to **`amI`**

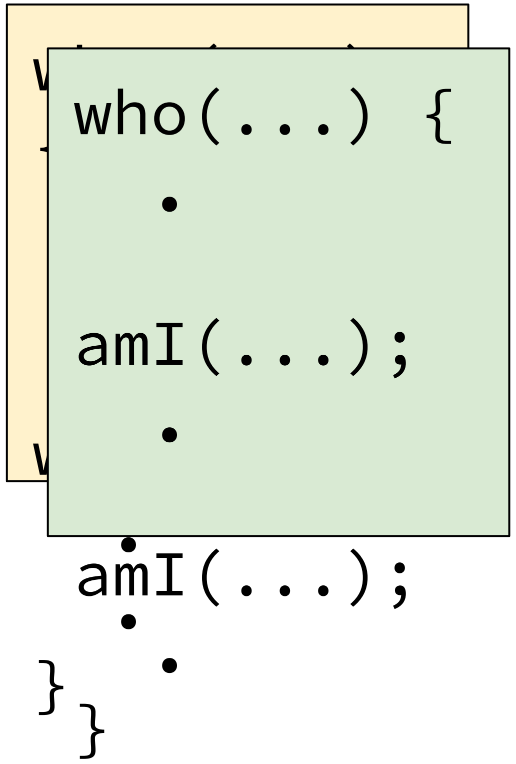


```
amI(...) {
   •
   •
   if(...)

amI(...);
   •
   •
}
```

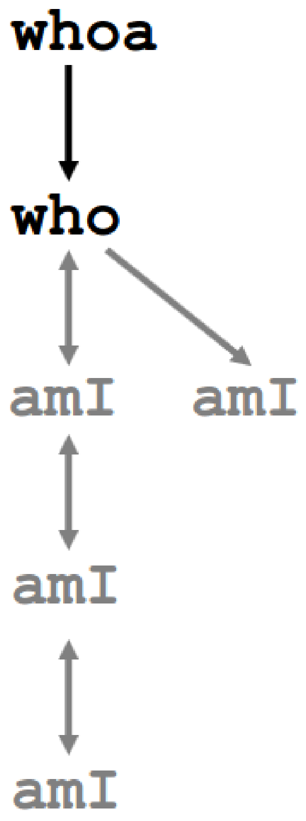


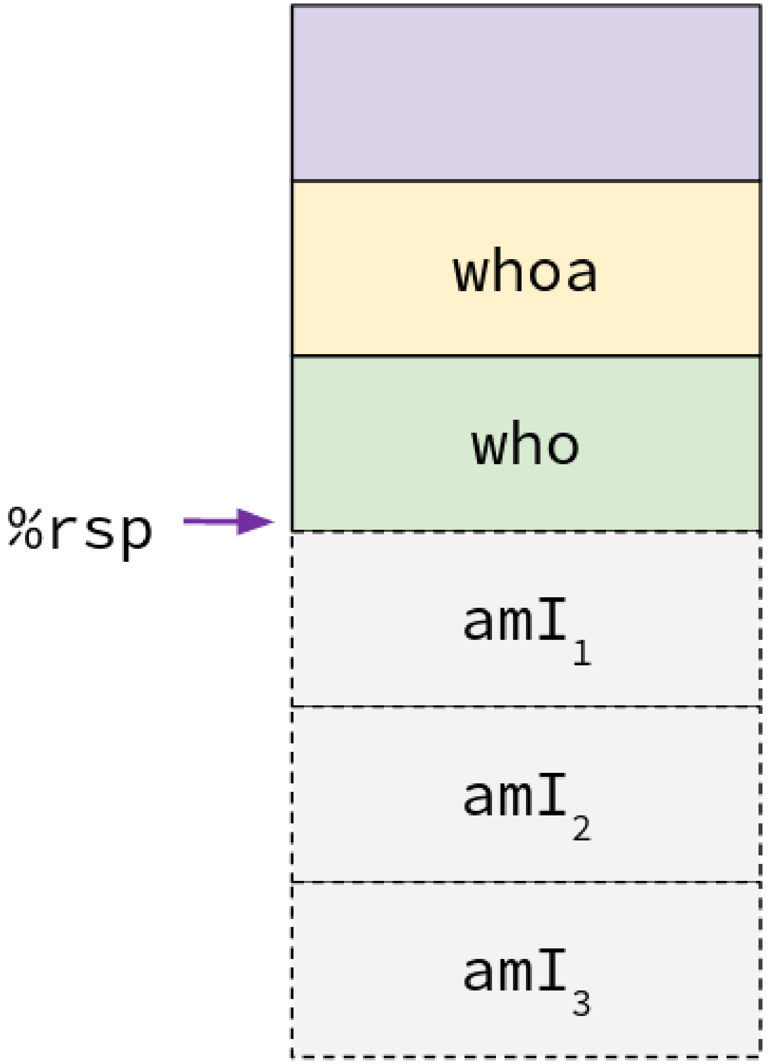


8. Return from Call to `amI`

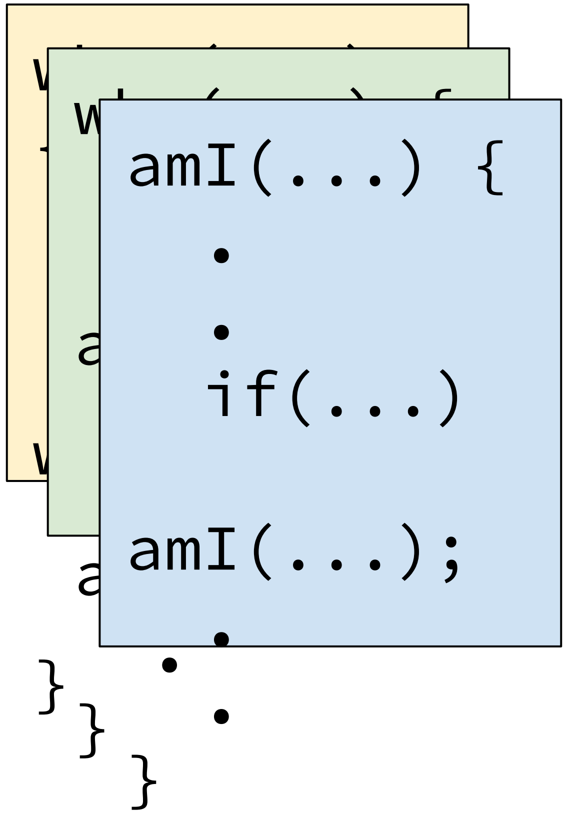


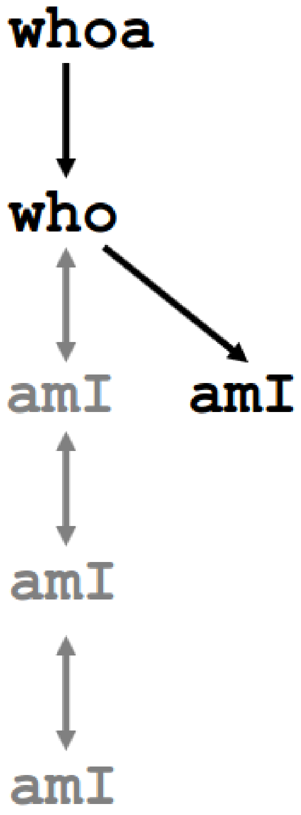
```
who(...) {
   •
   •
amI(...);
   •
   •
   •
amI(...);
   •
   •
}
```

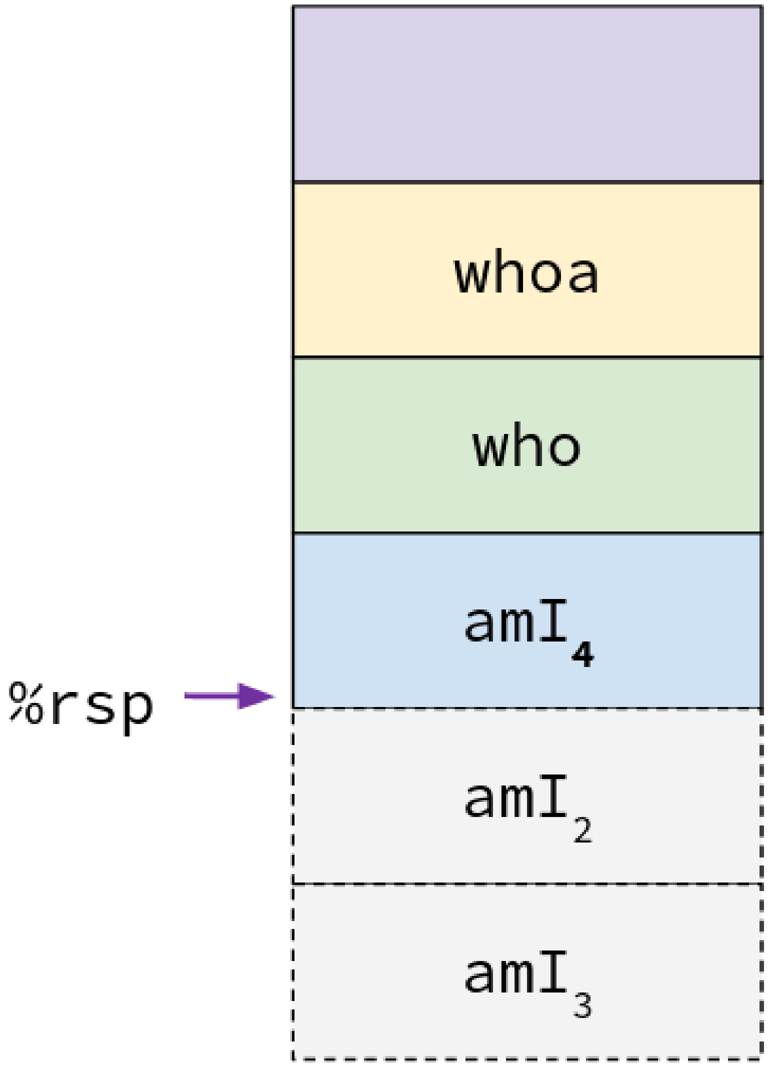




9. (yet another) Call to **`amI`**

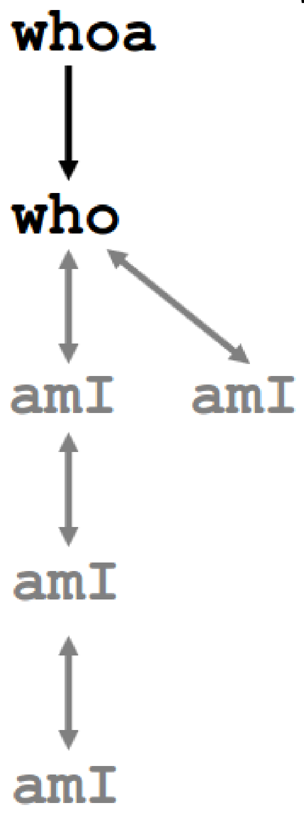

```
amI(...) {
  •
  •
  if(...)

amI(...);
  •
  •
}
```

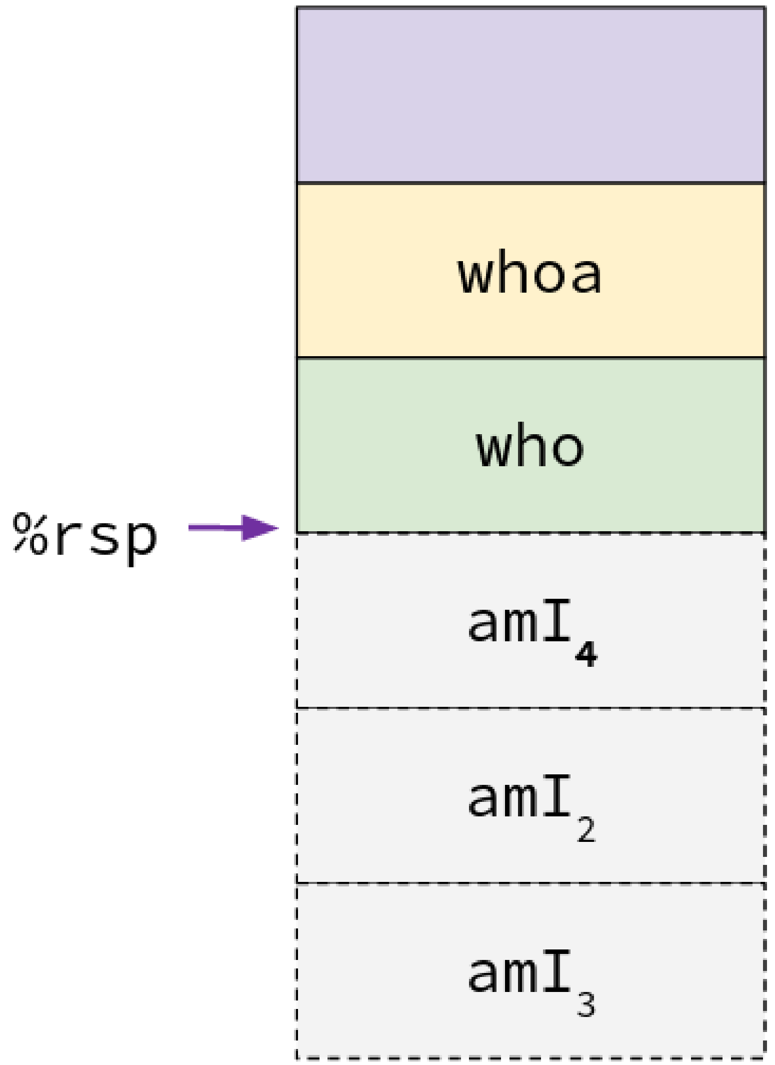




10. Return from (yet another) Call to amI

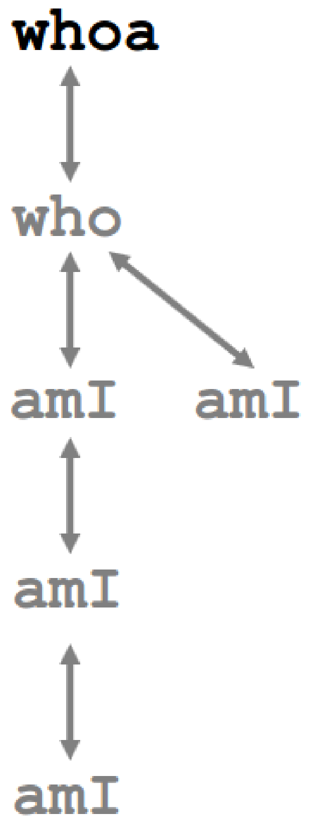
```
who(...) {
   •
 amI(...);
   •
 amI(...);
   •
}
```

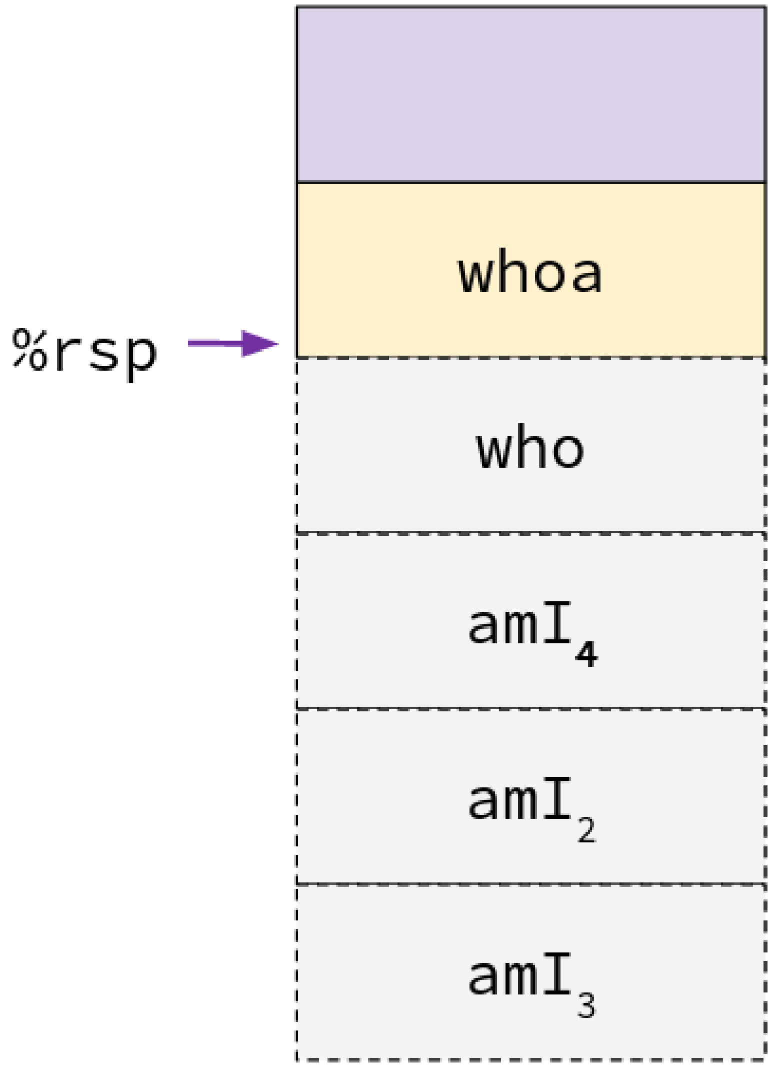




11. Return from Call to who

```
whoa(...)
{
   •
   •

who(...);
   •
   •
}
```





Stack Overflow

- When the the size of the stack grows too large
  - `%rsp` points to something it’s not supposed to, segmentation fault
  - In theory, happens when stack collides with heap
  - In practice, Linux limits stack to 8 MiB
- Aside: Stack Overflow website was named by popular vote from users. Some of the non-winning options:
  - bitoriented
  - dereferenced
  - privatevoid
  - shiftleft1
  - understandrecursion

Summary

- The **stack** is a region of memory that stores local data for **procedures**
    - Allocated in **frames**
    - Grows *down*. **Stack Pointer** (`%rsp`) points to the end of the stack
- When a procedure is called, **return address** is pushed onto the stack
    - Popped off again on return
- We use **procedure call convention** to pass data between procedures
    - 1st 6 args in registers (remember with **Di**ane's **si**lk **d**ress **c**osts $**89**)
    - Remaining args on the stack
    - Return value in %rax
- When writing to a register, save its old value on the stack to prevent data loss