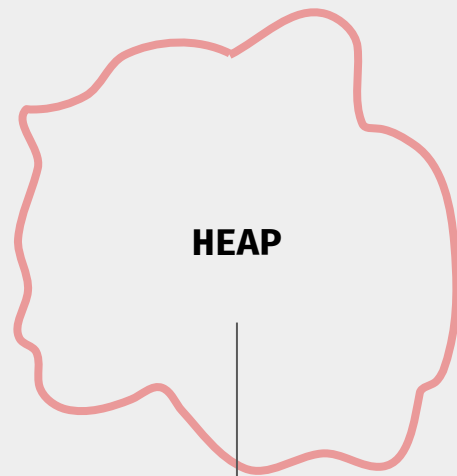


**OX** **BARE**  
**CB** **METAL**

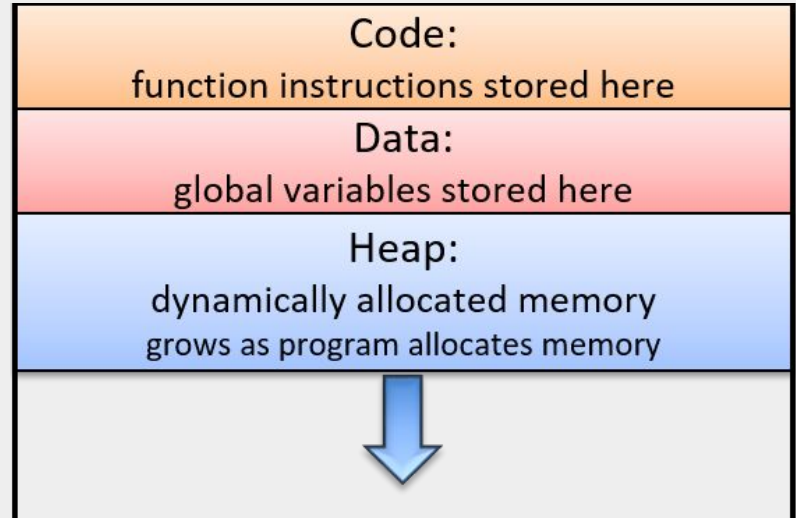
# Memory and the heap



0x200...	0x200...	0x200...	0x200...
0x200...	0x200...	0x200...	0x200...
0x200...	0x200...	0x200...	0x200...
0x200...	0x200...	0x200...	0x200...

# The Heap

Grows every time we  
create a .fill and  
STR/H/B data



STACK
0xdef45ea5
0x134a48fd
0x6785efae

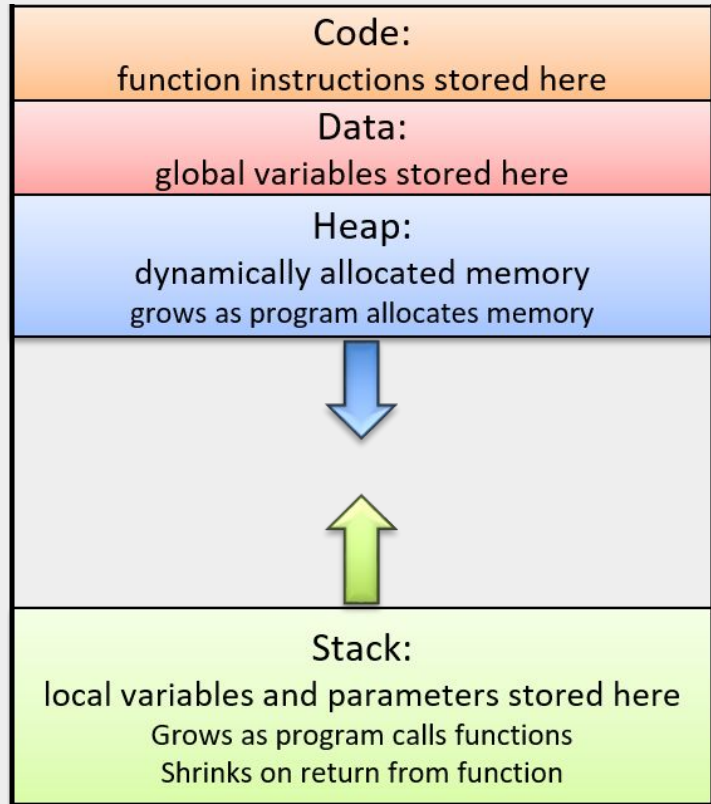
↑  
Lawful good



HEAP

↑  
Chaotic neutral

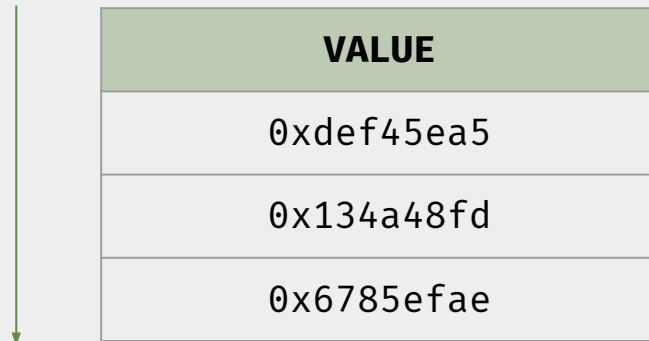
# Stack and Heap



# Brief intro the the Stack

LIFO

Last In, First Out



A diagram illustrating the stack's growth direction. A vertical green arrow points downwards from the top of a table to the text "grows down". The table has a header row labeled "VALUE" and three data rows containing hexadecimal values. The values are 0xdef45ea5, 0x134a48fd, and 0x6785efae, listed from top to bottom.

VALUE
0xdef45ea5
0x134a48fd
0x6785efae

“grows down”

# Brief intro the the Stack

- A linear data structure
- Largely used at the machine level to:
  - Keep track of function and subroutine addresses
  - Organize and keep track of function and subroutine arguments

# Using the Stack

PUSH: fills the stack with a singular value or set of values

REGISTERS	
R0	0xdef45ea5
R1	0x134a48fd
R2	0x6785efae

PUSH {R0-R2}

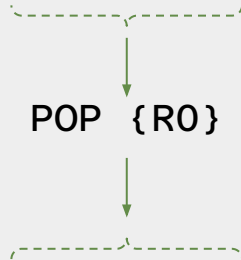
STACK
0xdef45ea5
0x134a48fd
0x6785efae



# Using the Stack

PUSH: Retrieves a single value or set of values

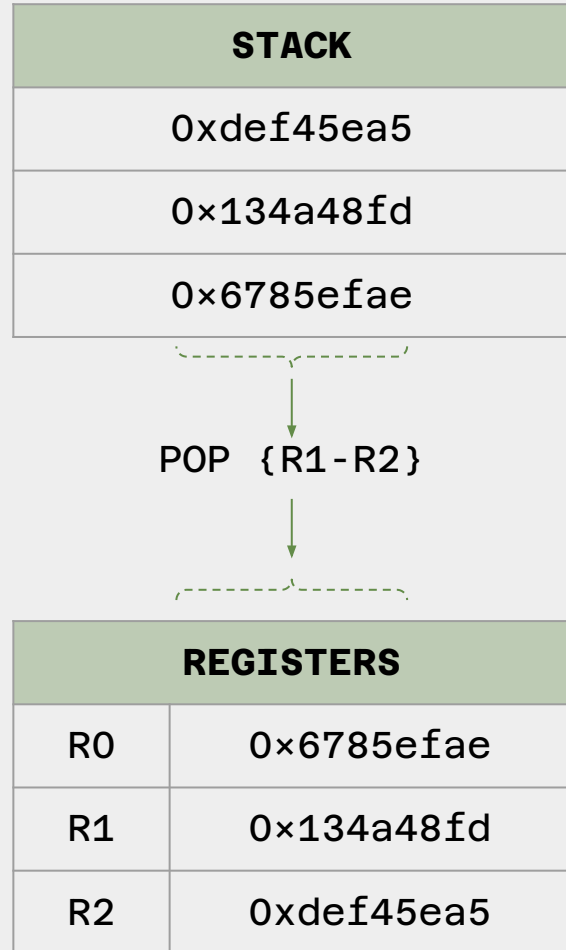
STACK
0xdef45ea5
0x134a48fd
0x6785efae



REGISTERS	
R0	0x6785efae
R1	
R2	

# Using the Stack

POP: Retrieves a single value or set of values



# PUSH and POP

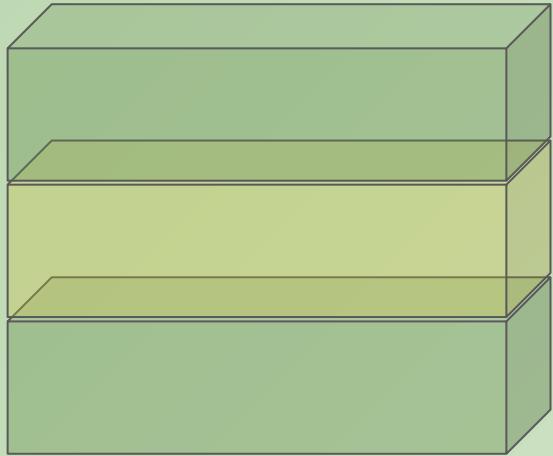
Argument formulae

PUSH  $\{R_M, R_D, \dots\}$

PUSH  $\{R_M - R_D\}$

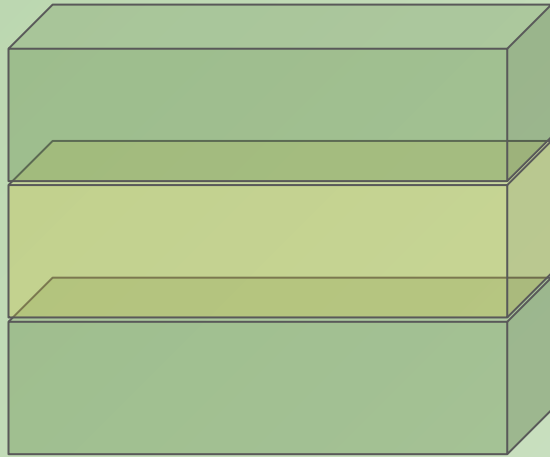
POP  $\{R_M, R_D, \dots\}$

POP  $\{R_M - R_D\}$



Individual stack “frame”

Contains all PUSH'd,  
POP'd values for a  
given subroutine



Implies: each subroutine  
can have its own stack  
“frame”

```
def add(a: int, b: int) -> int:
```

```
    c = a + b
```

```
    return c
```

```
s = add(2, 3)
```

```
print(s)
```

Program location  
Space for s

Program location  
Space for a  
Space for b

```
def add(a: int, b: int) -> int:
```

```
    c = a + b
```

```
    return c
```

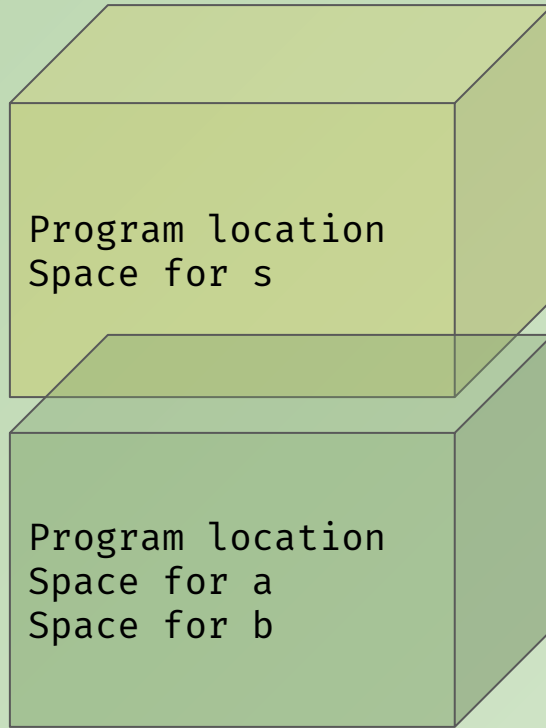
But we can't do  
this, can we?

```
s = add(2, 3)
```

```
print(s)
```

Program location  
Space for s

Program location  
Space for a  
Space for b

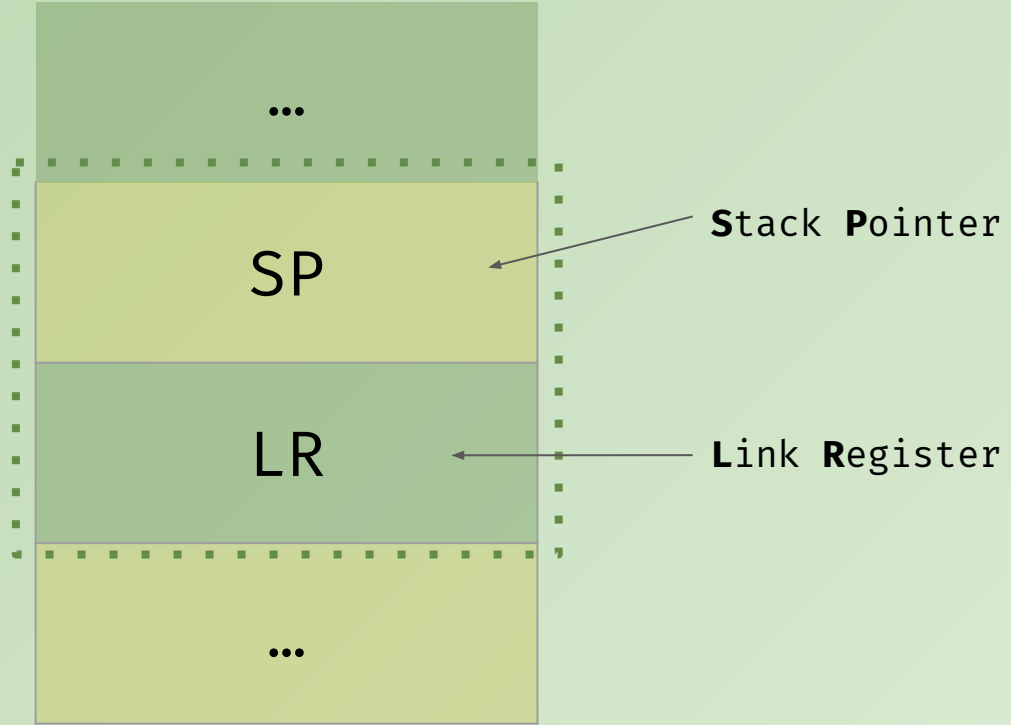
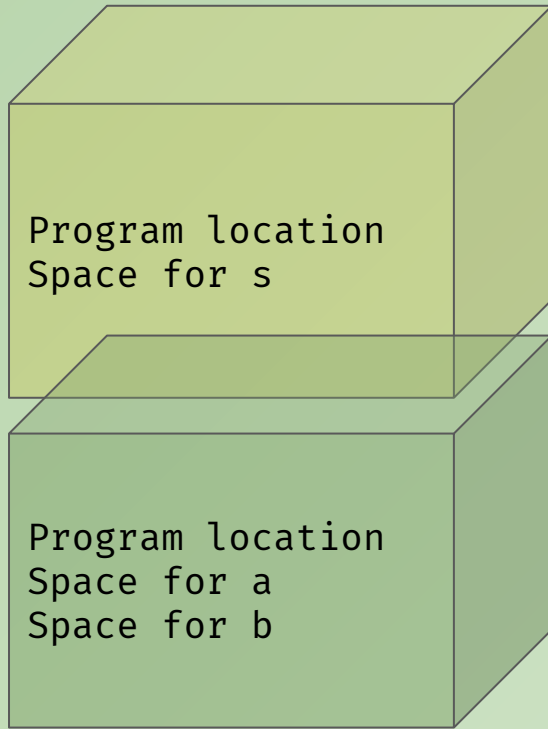


Back to here?



How do we get  
from here...





BL	<b>B</b> branch and <b>L</b> ink	Jump to a label; store location in <b>LR</b>
BX	<b>B</b> branch and <b>eX</b> ecute	Jump back to a memory location and continue to execute

**BL, BX**

BL

LABEL

BX

$R_D$