C Functions

CS 210 - Fall 2023
Vasiliki Kalavri
vkalavri@bu.edu

Defining and Calling Functions

- Before we go over the formal rules for defining a function, let’s look at our simple calculator program and how it defines functions.

return type

copy

```c
// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf("Result: %d\n", res);
}

// multiplies two numbers
int mul(int i, int j) {
  int k, res = 0;
  for (k=0; k<j; k++) {
    res += i;
  }
  printf("Result: %d\n", res);
  return res;
}
```

Return type

```c
// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf(“Result: %d\n", res);
}

// multiplies two numbers
int mul(int i, int j) {
  int k, res = 0;
  for (k=0; k<j; k++) {
    res += i;
  }
  printf("Result: %d\n", res);
  return res;
}
```

Return type

```c
// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf(“Result: %d\n", res);
}

// multiplies two numbers
int mul(int i, int j) {
  int k, res = 0;
  for (k=0; k<j; k++) {
    res += i;
  }
  printf("Result: %d\n", res);
  return res;
}
```

Parameters

```
int main() {
  int i = 2;
  power(2, i);
  return 0;
}

// computes i^j using mul()
int power(int i, int j) {
  int res = 1;
  while(j > 0) {
    res = mul(res, i);
    j--;
  }
  return res;
}
```

```
int main() {
  int i = 2;
  power(2, i);
  return 0;
}

// computes i^j using mul()
int power(int i, int j) {
  int res = 1;
  while(j > 0) {
    res = mul(res, i);
    j--;
  }
  return res;
}
```

Arguments are passed *by value*

- When a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument.

```c
int main() {
  int i = 2;
  power(2, i);
  return 0;
}

// computes i^j using mul()
int power(int i, int j) {
  int res = 1;
  while(j > 0) {
    res = mul(res, i);
    j--;
  }
  return res;
}
```

Arguments are passed *by value*

Function Declarations

- C doesn’t require that the definition of a function precede its calls.

```c
#include <stdio.h>

int main() {

  int i, j;

  printf("Enter two numbers: ");
  scanf("%d %d", &i, &j);
  add(i, j);
  return 0;
}

// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf("Result: %d\n", res);
}
```

compile fine

Function Declarations

- C doesn't require that the definition of a function precede its calls.

```c
#include <stdio.h>

int main() {

  int i, j;

  printf("Enter two numbers: ");
  scanf("%d %d", &i, &j);
  add(i, j);
  return 0;
}

// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf("Result: %d\n", res);
}
```

```
$ gcc -fno-inline -fno-stack-protector -fno-pic -static -fcf-protection=none -fno-asynchronous-unwind-tables -Os -masm=intel -g add.c -o add
add.c: In function 'main':
add.c:7:3: warning: implicit declaration of function 'add' [-Wimplicit-function-declaration]
    7 |   add(i, j);
      |   ^~~
add.c: At top level:
add.c:12:6: warning: conflicting types for 'add'
   12 | void add(int i, int j) {
      |      ^~~
add.c:7:3: note: previous implicit declaration of 'add' was here
    7 |   add(i, j);
      |   ^~~
```

Function Declarations

- When the compiler encounters the first call of `add` in `main`, it has no information about the function.
- Instead of producing an error message, the compiler assumes that `add` returns an `int` value.
- The compiler has created an ***implicit declaration*** of the function.

Function Declarations

- A better solution: declare each function before calling it.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  return-type function-name (*parameters*) ;

- The declaration of a function must be consistent with the function's definition.

```c
/**
* A basic integer calculator.
* It supports addition, subtraction,
* multiplication, division, and power.
*
* **/

#include <stdio.h>
#include <stdlib.h>

void print_help();
void add(int, int);
int mul(int, int);
int division(int i, int j);
int power(int i, int j);
int rec_power(int i, int j);
```

```c
int main() {

  int i, j;
  char op;

  // print instructions
  print_help();

  while (1) {
    printf("Enter operator or Q to exit: ");
    // read operation
    scanf(" %c ", &op);

    if (op != 'Q') {
      // read input numbers
      printf("Enter two integers: ");
      scanf("%d %d", &i, &j);

      if (i < 0 || j < 0) {
        printf("Negative numbers are ignored.\n");
        continue;
      }

      // apply operation
      switch (op) {
        case '+': add(i, j); break;
        case '-': printf("Result: %d\n", i-j); break;
        case '*': mul(i, j); break;
        case '/': {
          int ret = division(i, j);
          if (ret < 0) break;
          printf("Result: %d\n", ret);
          break;
        }
        case '^': printf("Result: %d\n", power(i, j)); break;
      default:  printf("Invalid operator.\n"); break;
      }
    }
    else {
      printf("Sorry to see you go :/\n");
      return 0;
    }
  }
  return 0;
}
```

```c
int main() {

  int i, j;
  char op;

  // print instructions
  print_help();

  while (1) {
    printf("Enter operator or Q to exit: ");
    // read operation
    scanf(" %c ", &op);

    if (op != 'Q') {
      // read input numbers
      printf("Enter two integers: ");
      scanf("%d %d", &i, &j);

      if (i < 0 || j < 0) {
        printf("Negative numbers are ignored.\n");
        continue;
      }

      // apply operation
      switch (op) {
        case '+': add(i, j); break;
        case '-': printf("Result: %d\n", i-j); break;
        case '*': mul(i, j); break;
        case '/': {
          int ret = division(i, j);
          if (ret < 0) break;
          printf("Result: %d\n", ret);
          break;
        }
        case '^': printf("Result: %d\n", power(i, j)); break;
      default:  printf("Invalid operator.\n"); break;
      }
    }
    else {
      printf("Sorry to see you go :/\n");
      return 0;
    }
  }
  return 0;
}
```

```c
/** Prints the calculator usage instructions **/
void print_help() {
  printf("\n***Welcome to the basic calculator program!***\n\n");
  printf("Select one of the following operations or type 'Q' to exit\n");
  printf("+\t: Addition\n");
  printf("-\t: Subtraction\n");
  printf("*\t: Multiplication\n");
  printf("\\\t: Division\n");
  printf("^\t: Power\n");
}

// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf("Result: %d\n", res);
}

// multiplies two numbers
int mul(int i, int j) {
  int k, res = 0;
  for (k=0; k<j; k++) {
    res += i;
  }
  printf("Result: %d\n", res);
  return res;
}
```

Attendance

Opcodes and C

When we write assembly code we are free to layout our opcodes and use registers in any way we like.

- We can place labels anywhere in our opcodes.
- We can specify a jump to any arbitrary location.
- While we can use processor support for passing return address via instructions, like `call` and `ret`, we are not required too.
- We are not forced to use the registers in any particular way.

C Standardizes how to organize and write opcodes

C forces us to decompose and organize opcodes into “functions”

- global label - single entry point
- block of opcodes ending in a “return”
- standardizes use of registers
- standardizes use of stack use of stack allocate at the memory
    - call frames - automatic storage of locals
    - separation into declaration (many) and definition (one)
    - compiler can get smart and optimize functions and variables away
        - in-line : technique that eliminate jump and return
        - dead-code elimination : compilar can figure out any codes that never called. remove it from assembly
        - register only variables

Let’s take a look at the assembly code to see how C works

```
void myfunc(void) {}
```

returns noting, noting as a parameter,

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel
myfunc0.c -o myfunc0.s
```

compile

Let's take a look at the assembly code to see how C works

```
void myfunc(void) {}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc0.c -o myfunc0.s
```

```
    .file  "myfunc0.c"
    .intel_syntax noprefix
    .text
    .globl myfunc
    .type  myfunc, @function
myfunc:
    ret
    .size  myfunc, .-myfunc
    .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
    .section .note.GNU-stack,"",@progbits
```

→ Compiler create global Entry.

Calling a function

```
__attribute__ ((noinline))
```

2functions

A
```
int funcA(void) {
  return 7;
}
```

B
```
int funcB(void) {
  return 3 + funcA();
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc1.c -o myfunc1.s
```

eax : return

35:29

Calling a function

```
__attribute__ ((noinline))

int funcA(void) {
  return 7;
}

int funcB(void) {
  return 3 + funcA();
}
```

```
 .file "myfunc1.c"
  .intel_syntax noprefix
  .text
  .globl funcA
  .type  funcA, @function
funcA:
  mov  eax, 7
  ret
  .size  funcA, .-funcA
  .globl funcB
  .type  funcB, @function
funcB:
  call funcA
  add  eax, 3
  ret
  .size  funcB, .-funcB
  .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
  .section .note.GNU-stack,"",@progbits
```

eax uses as return value.

function A return 7

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc1.c -o myfunc1.s
```

Passing arguments

```
__attribute__ ((noinline))

int func2(int x, int y) {
  return x + y;
}

int func1(int x) {
  return func2(x,2);
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel
myfunc3.c -o myfunc3.s
```

Passing arguments

```c
__attribute__ ((noinline))

int func2(int x, int y) {
  return x + y;
}

int func1(int x) {
  return func2(x,2);
}
```

```
    .file  "myfunc3.c"
    .intel_syntax noprefix
    .text
    .globl func2
    .type  func2, @function
func2:
    lea  eax, [rdi+rsi]
    ret
    .size  func2, .-func2
    .globl func1
    .type  func1, @function
func1:
    mov  esi, 2
    jmp  func2
    .size  func1, .-func1
    .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
    .section .note.GNU-stack,"",@progbits
```

just use calculation.

load effective address

let compiler to do very fast memory computation.
Arithmetics operation

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel
myfunc3.c -o myfunc3.s
```

Passing arguments

```
__attribute__ ((noinline))

int func2(int x, int y) {
  return x + y;
}

int func1(int x) {
  return func2(x,2);
}
```

```
  .file  "myfunc3.c"
  .intel_syntax noprefix
  .text
  .globl func2
  .type  func2, @function
func2:
  lea  eax, [rdi+rsi]
  ret
  .size  func2, .-func2
  .globl func1
  .type  func1, @function
func1:
  mov  esi, 2
  jmp  func2
  .size  func1, .-func1
  .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
  .section .note.GNU-stack,"",@progbits
```

Why these registers?

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc3.c -o myfunc3.s
```

A fixed way for passing arguments

- **First 6 arguments** will be assigned to these registers

`rdi`
`rsi`
`rdx`
`rcx`
`r8`
`r9`

more than 6th Argument will be located in stack

- **Return value**

`rax`

• • •
Arg *n*
• • •
Arg 8
Arg 7

8
7

6th rdx
rdi
rcx
rsi
r8 8 9
r9

rax

- Only allocate stack space when needed

***D**iane **S**ips **D**elicious **C**offee **8** out of **9** times*

Diane Sips Delicious Coffee 8 out of 9 times.

Register saving conventions

- When `foo` calls `who`:
  - `foo` is the *caller* 부르는 function A
  - `who` is the *callee* 불리는 function B

caller is responsible for saving before the call (put into stack)

- Can a register be used for temporary storage?
- Conventions
  - *"Caller Saved"*
    - Caller saves temporary values in its frame before the call
  - *"Callee Saved"*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

Stack-based Languages

we need data structure to save callee, caller variable

" stack "

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single function
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

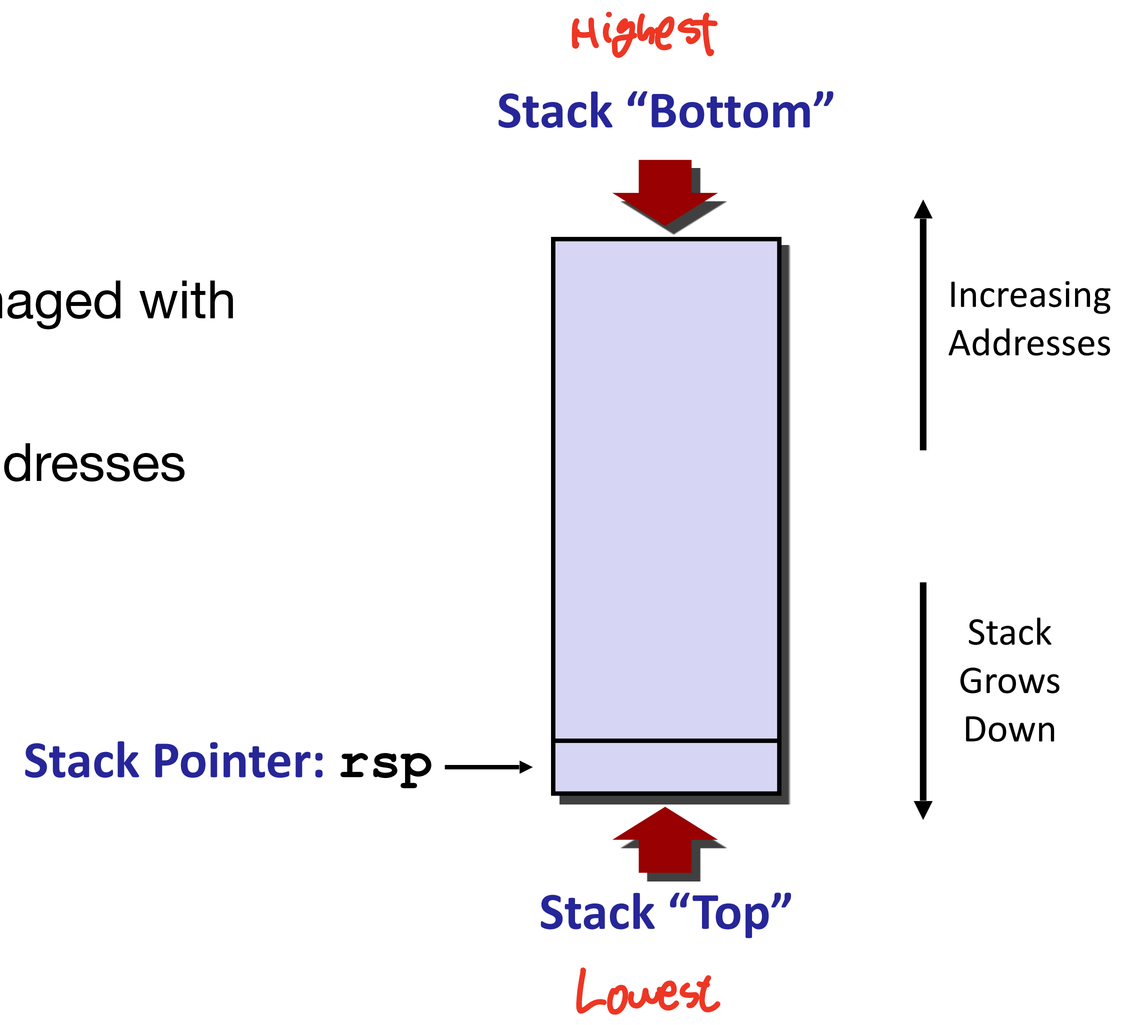
F → Q

Q returns before f

- Stack discipline
  - State for a function is needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in [*Frames*]
  - state for single function instantiation

x86-64 Stack

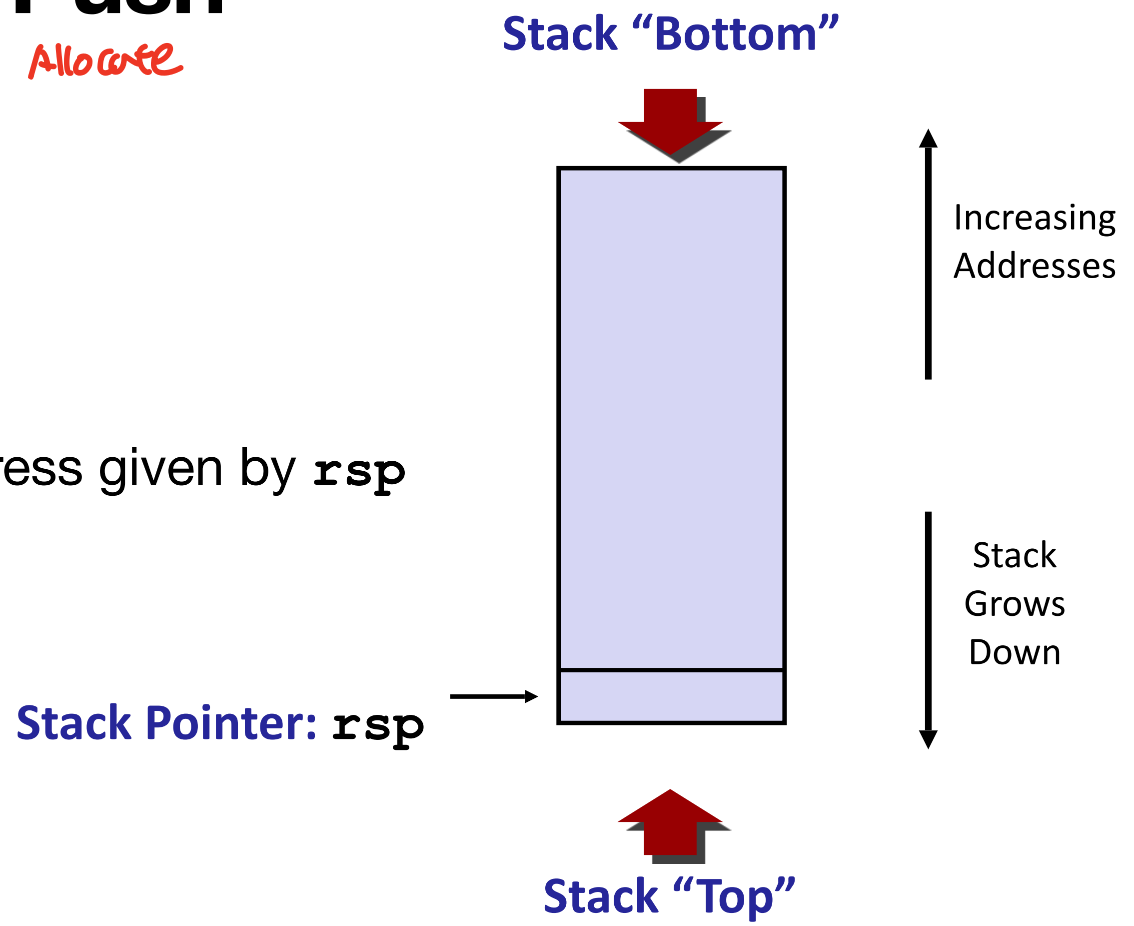
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `rsp` contains lowest stack address
  - address of “top” element



x86-64 Stack: Push

Allocate

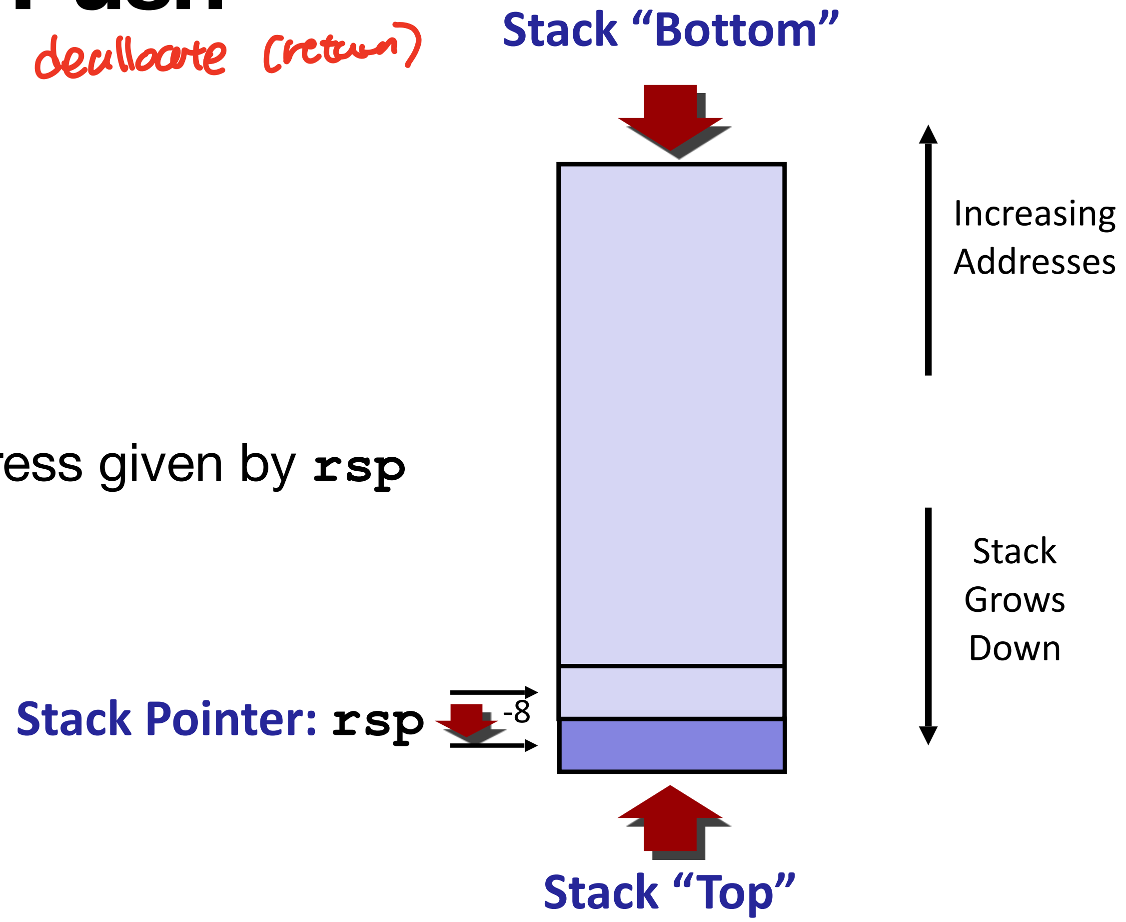
- push *Src*
  - Fetch operand at *Src*
  - Decrement **rsp** by 8
  - Write operand at address given by **rsp**



x86-64 Stack: Push

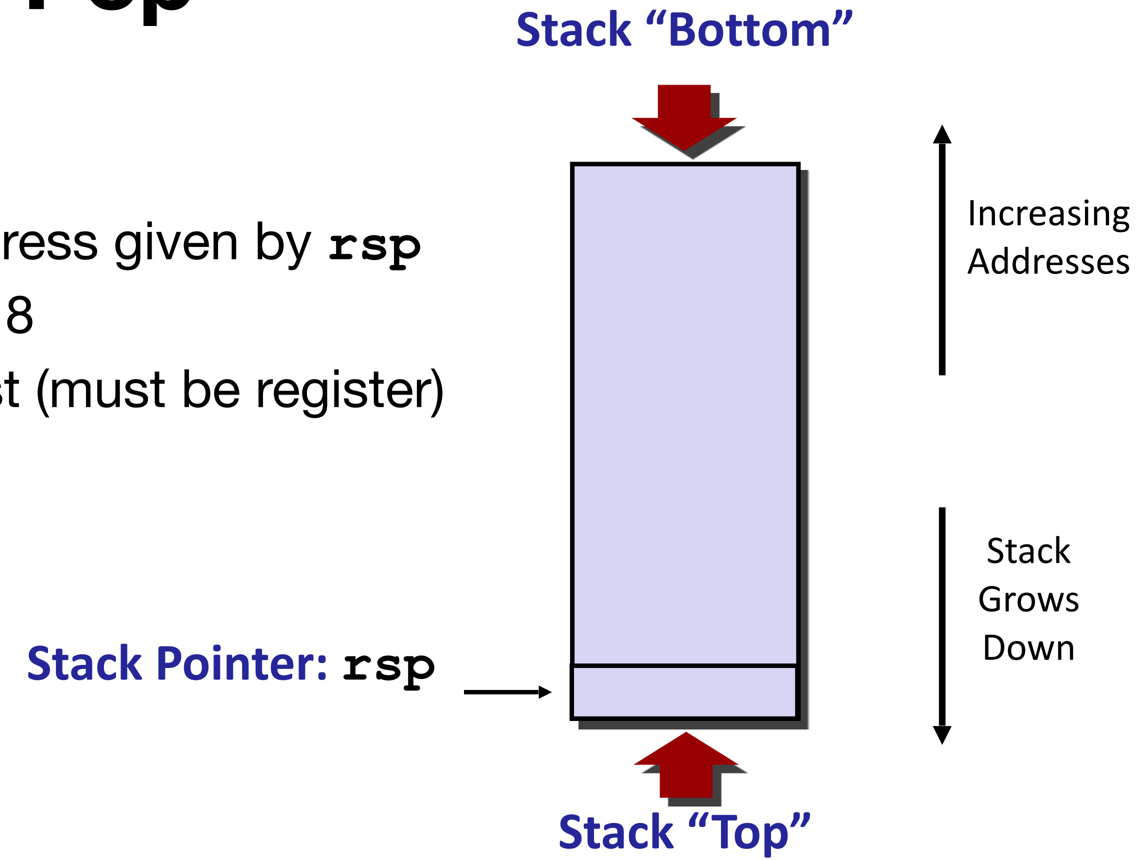
deallocate (return)

- push *Src*
  - Fetch operand at *Src*
  - Decrement **`rsp`** by 8
  - Write operand at address given by **`rsp`**



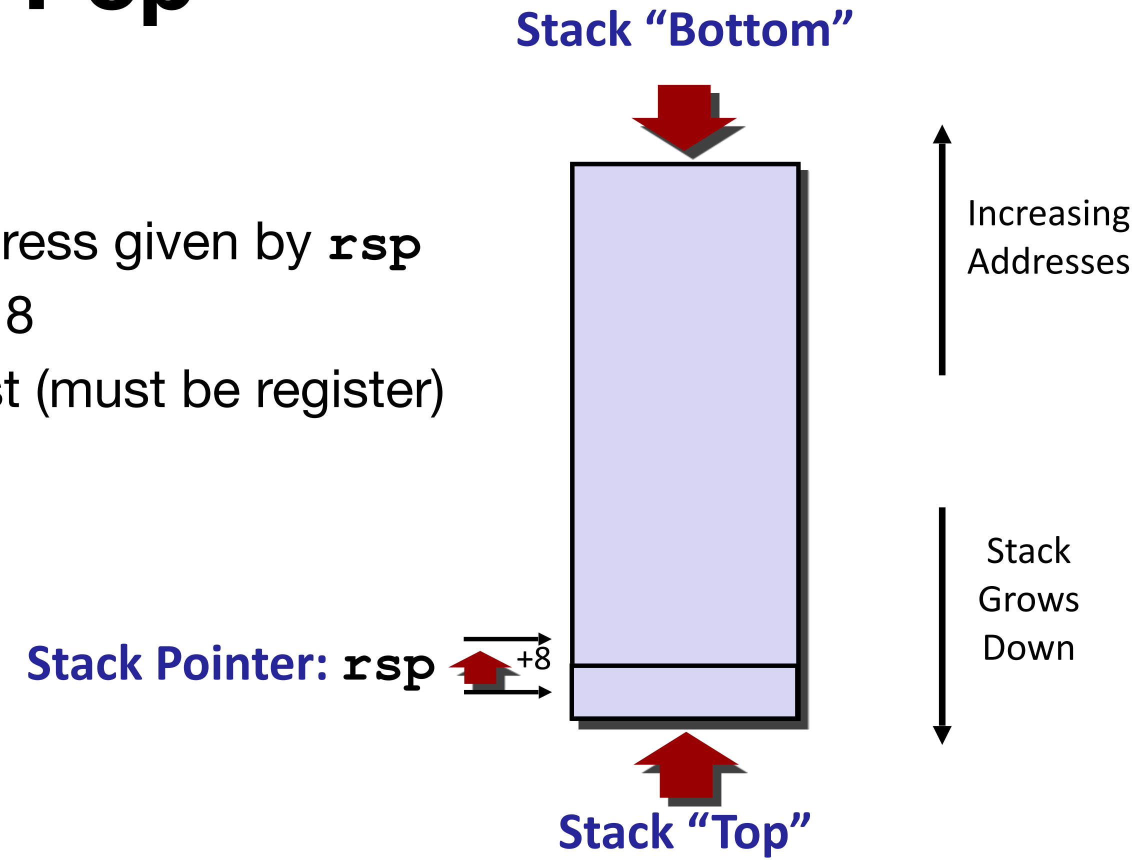
x86-64 Stack: Pop

- pop *Dest*
  - Read value at address given by **`rsp`**
  - Increment **`rsp`** by 8
  - Store value at Dest (must be register)



x86-64 Stack: Pop

- pop *Dest*
  - Read value at address given by **`rsp`**
  - Increment **`rsp`** by 8
  - Store value at Dest (must be register)



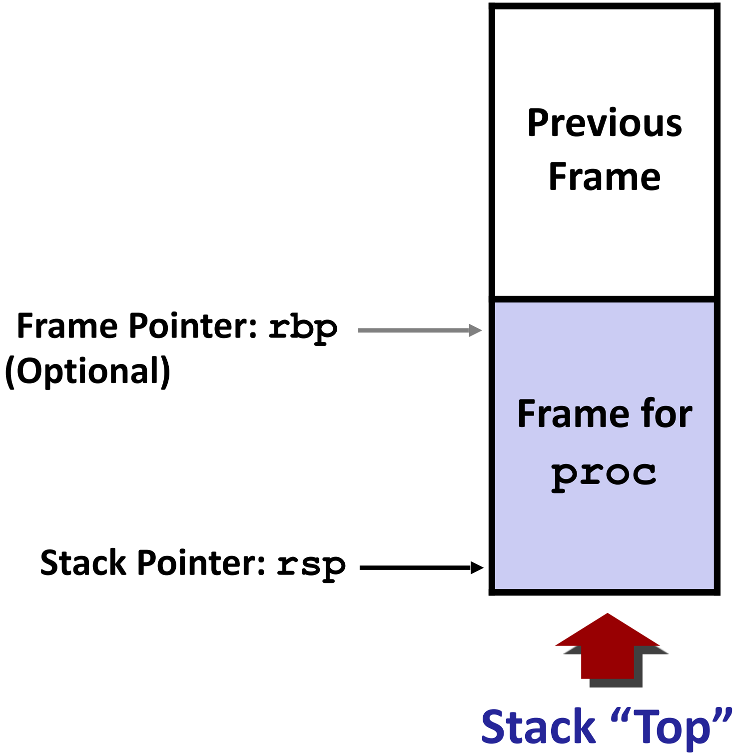
Stack Frames

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

- Management
  - Space allocated when entering the function
    - "Set-up" code
    - Includes push by **`call`** instruction

call includes push

  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction



Function Control Flow

- Use stack to support call and return

- Function call: `call label`
  - Push return address on stack
  - Jump to ***label***

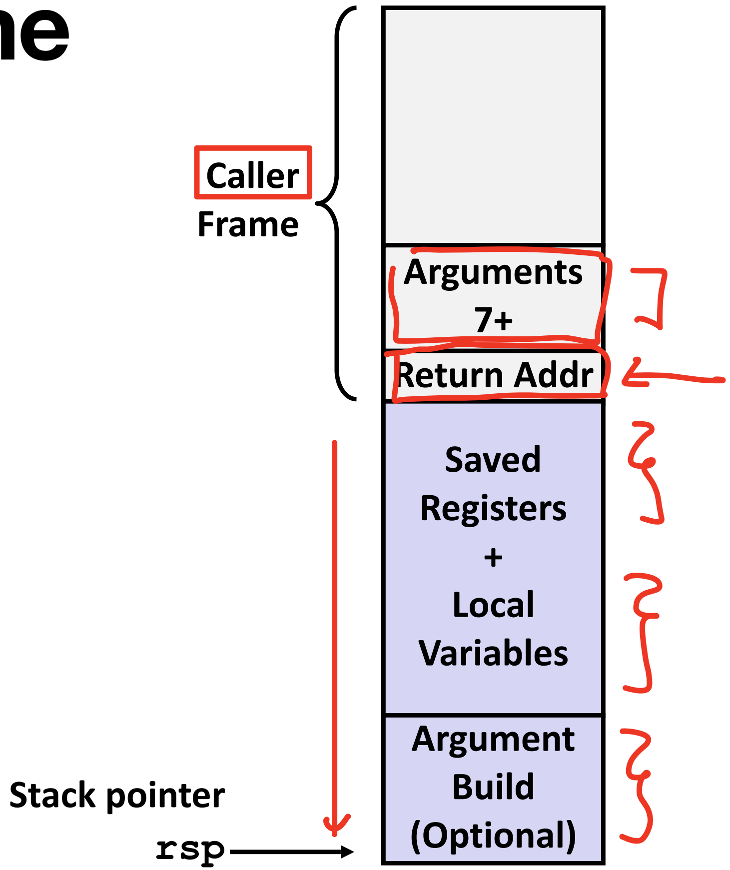
- Return address:
  - Address of the next instruction right after call

- Function return: `ret`
  - Pop address from stack
  - Jump to address

x86-64/Linux Stack Frame

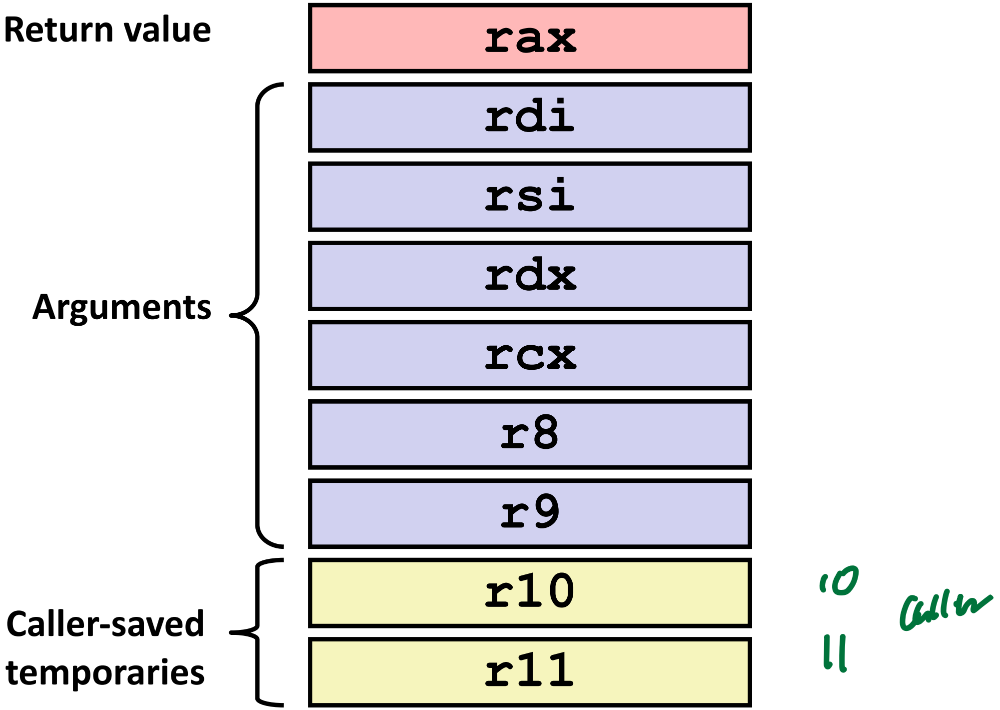
- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context

- Caller Stack Frame
  - Return address
    - Pushed by **`call`** instruction
  - Arguments for this call



x86-64 Linux Register Usage #1

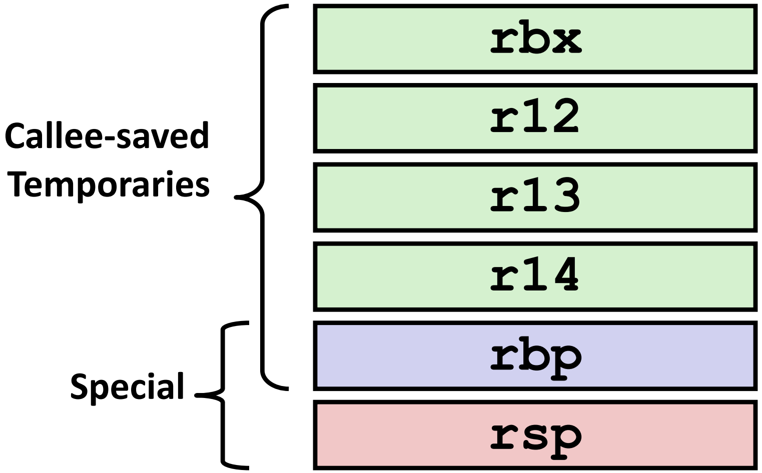
- `rax`
  - Return value
  - Also caller-saved
  - Can be modified by function
- `rdi, …, r9`
  - Arguments
  - Also caller-saved
  - Can be modified by function
- `r10, r11`
  - Caller-saved
  - Can be modified by function



x86-64 Linux Register Usage #2

rbx

- `rbx, r12, r13, r14`
  - Callee-saved
  - Callee must save & restore
- `rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `rsp`
  - Special form of callee save
  - Restored to original value upon exit from function



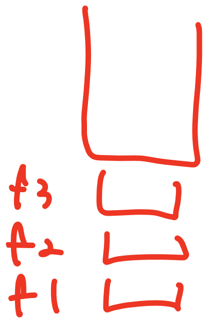
Recursion

- A function is ***recursive*** if it calls itself.
- The following function computes $n!$ recursively:

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

calling itself

Recursion



- To see how recursion works, let's trace the execution of the statement

  `i = fact(3);`

  `fact(3)` finds that 3 is not less than or equal to 1, so it calls
  `fact(2)`, which finds that 2 is not less than or equal to 1, so it calls
  `fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing
  `fact(2)` to return 2 × 1 = 2, causing
  `fact(3)` to return 3 × 2 = 6.

Call Chain Example

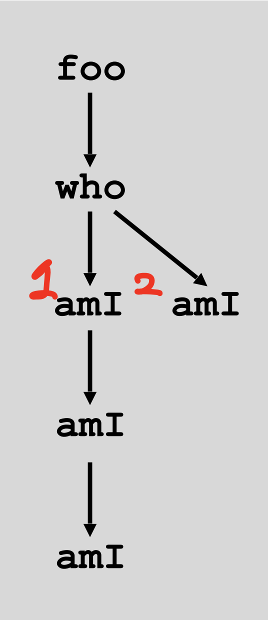
```
foo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
 1 amI();
   • • •
 2 amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

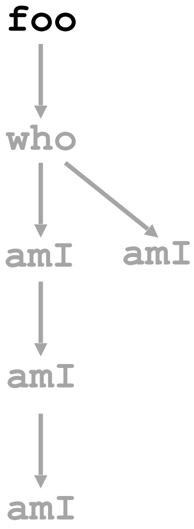
Procedure **`amI()`** is recursive

Example Call Chain

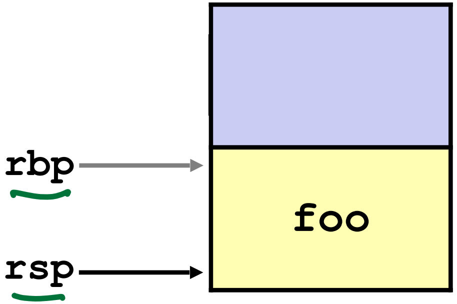


Example

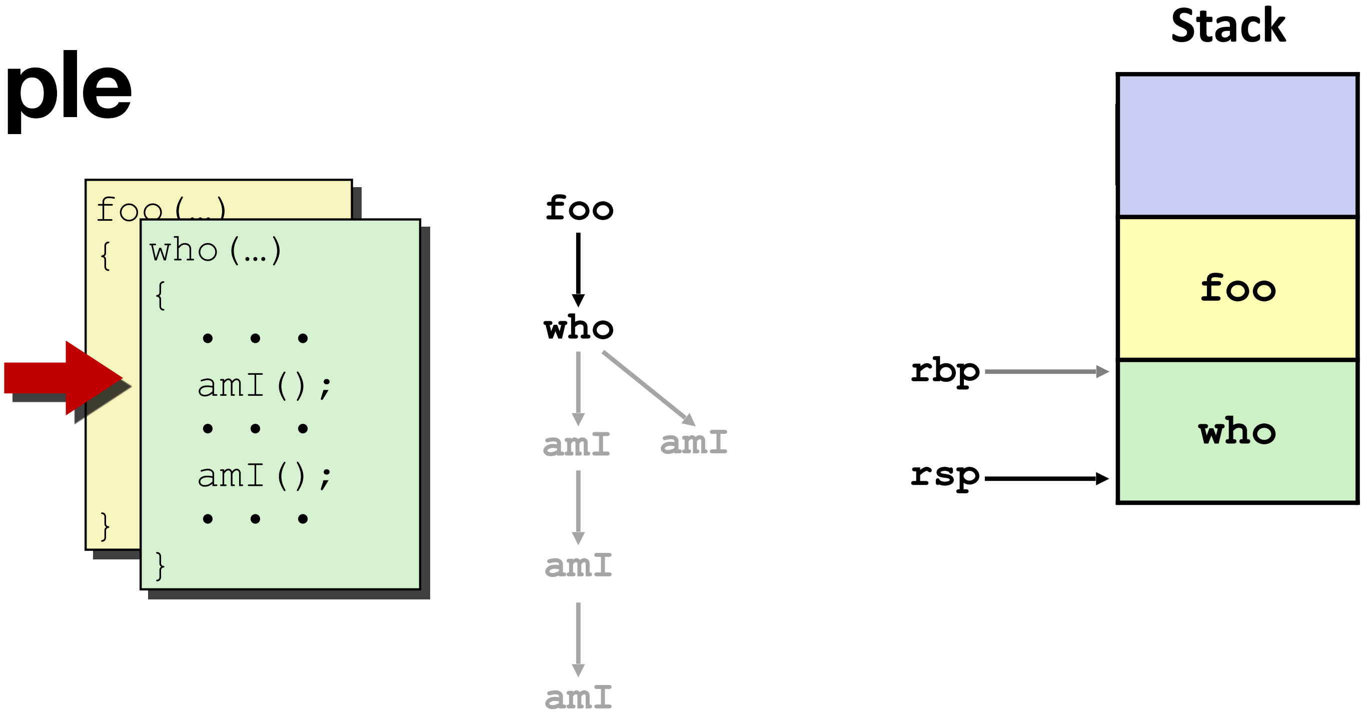
```
foo(…)
{
   •
   •
   who();
   •
   •
}
```



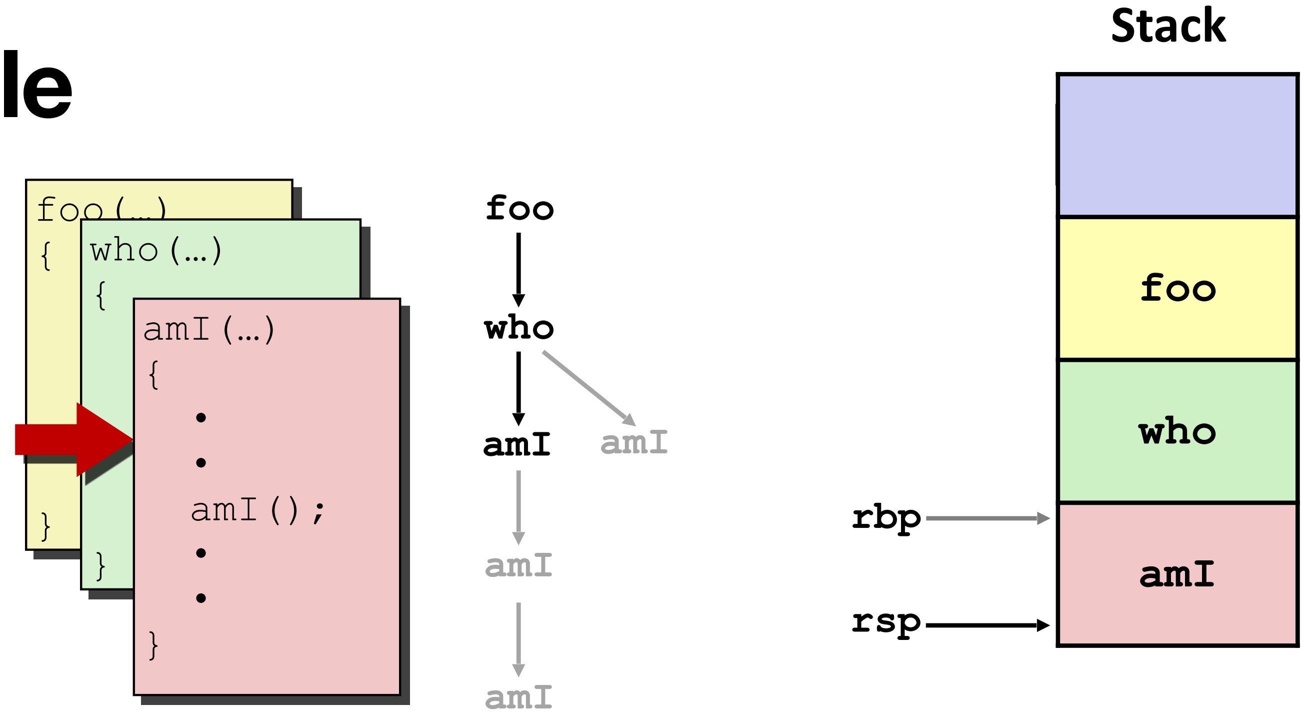
Stack



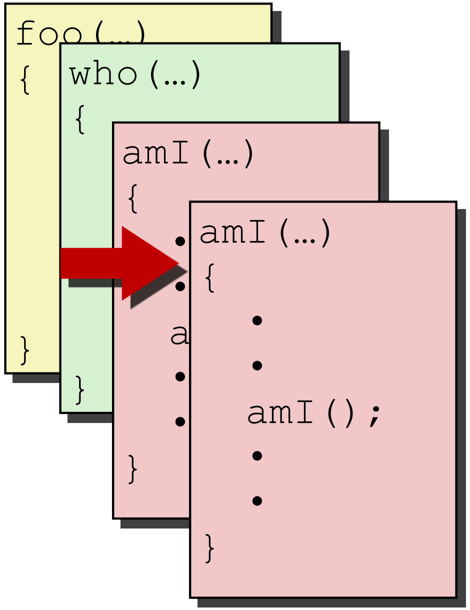
Example

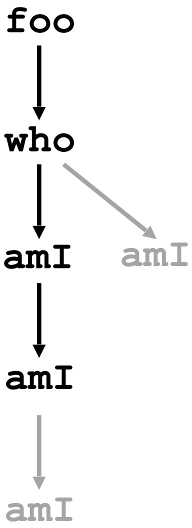


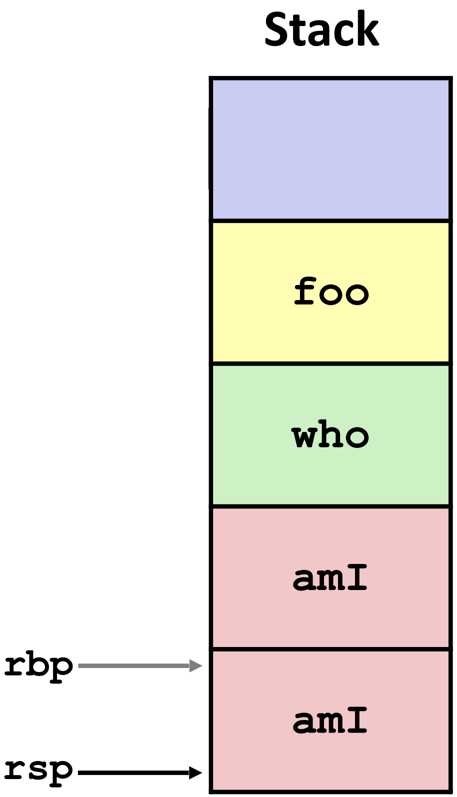
Example


foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
foo
who
amI
amI
amI
amI
Stack
foo
who
amI
rbp
rsp

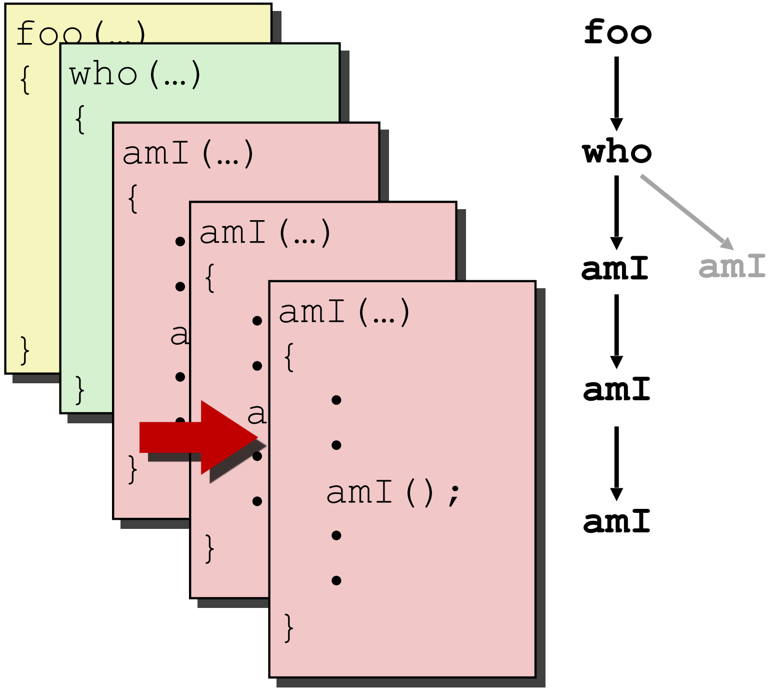
Example


foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}

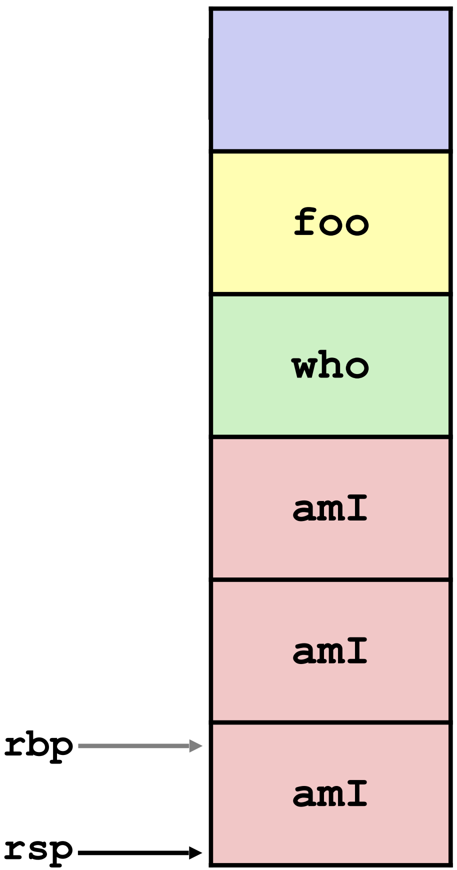

foo
who
amI
amI
amI
amI


Stack
foo
who
amI
amI
rbp
rsp

Example

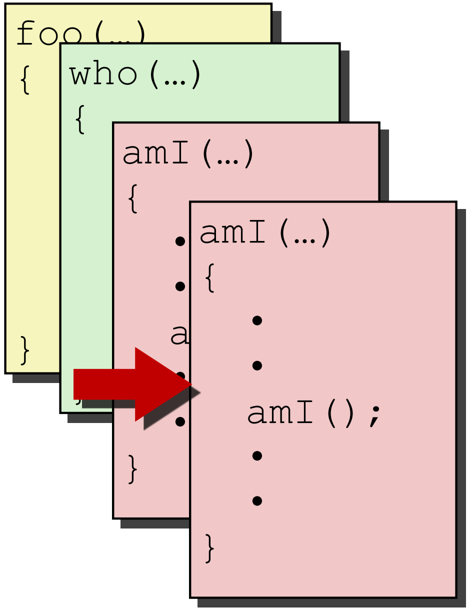


foo(…)
{
}
who(…)
{
}
amI(…)
{
a
}
amI(…)
{
a
}
amI(…)
{
amI();
}
foo
who
amI
amI
amI
amI

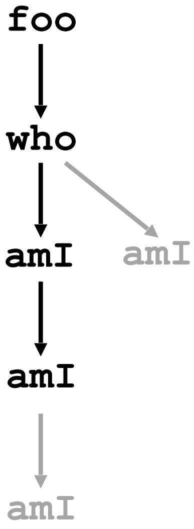


Stack
foo
who
amI
amI
amI
rbp
rsp

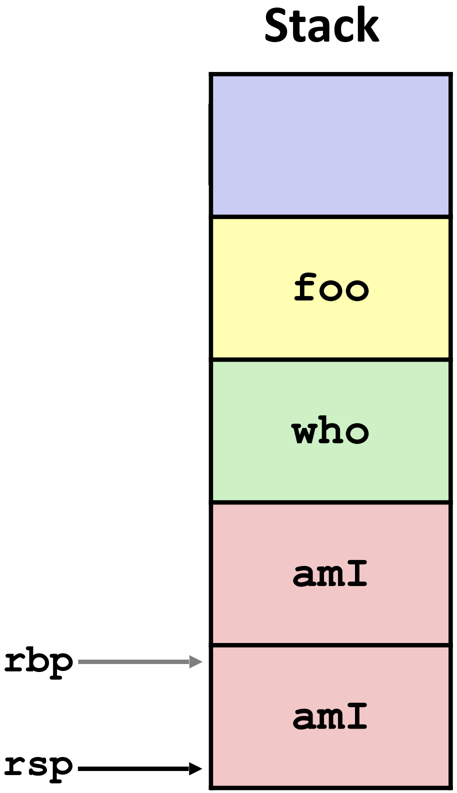
Example



foo(…)
{
}
who(…)
{
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
amI();
•
•
}

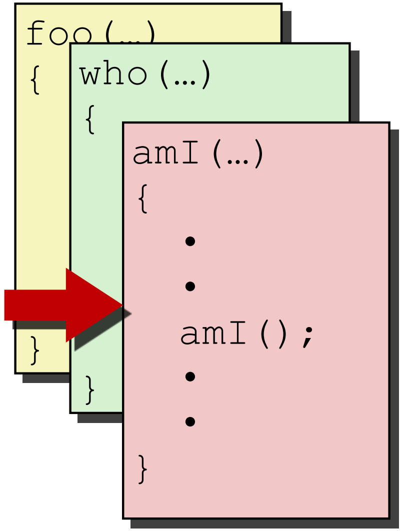


foo
who
amI
amI
amI
amI

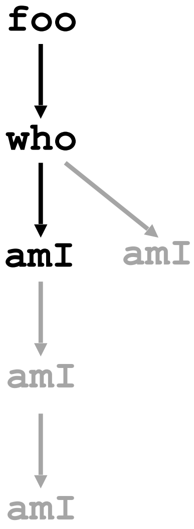


Stack
foo
who
amI
amI
rbp
rsp

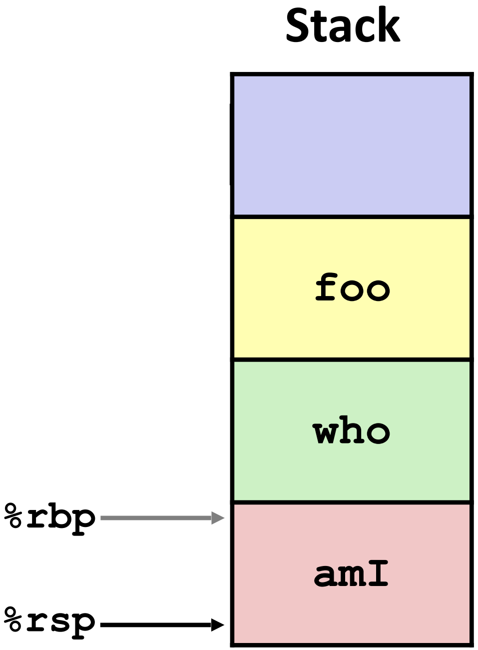
Example



foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}

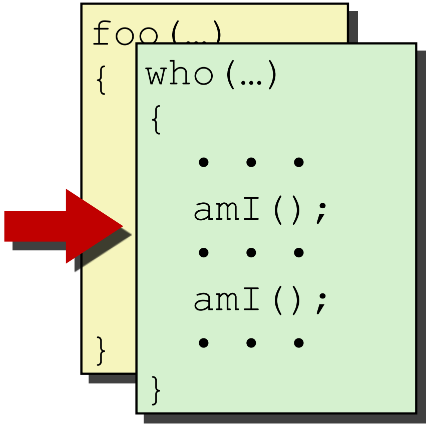


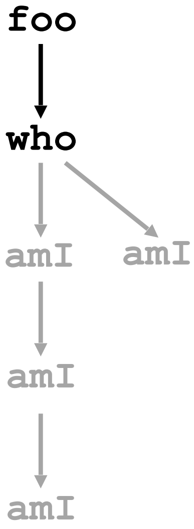
foo
who
amI
amI
amI
amI

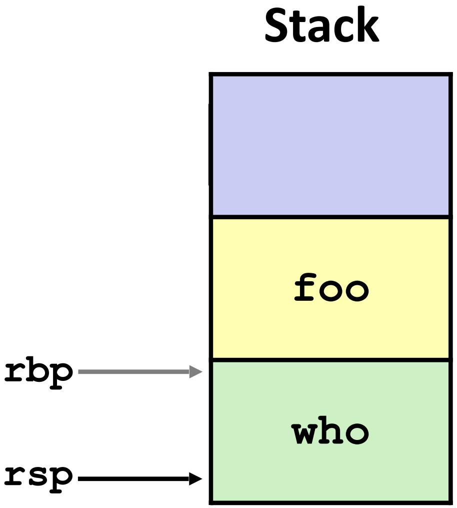


Stack
foo
who
amI
%rbp
%rsp

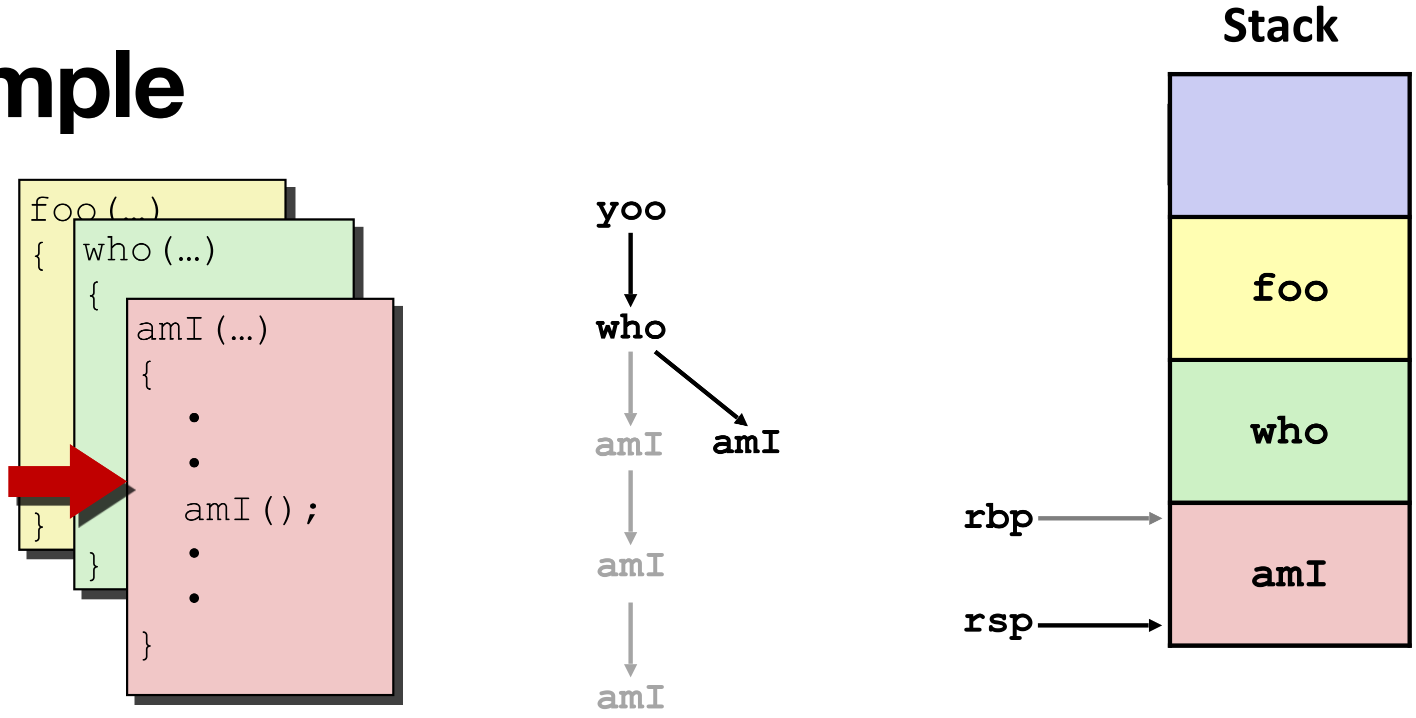
Example



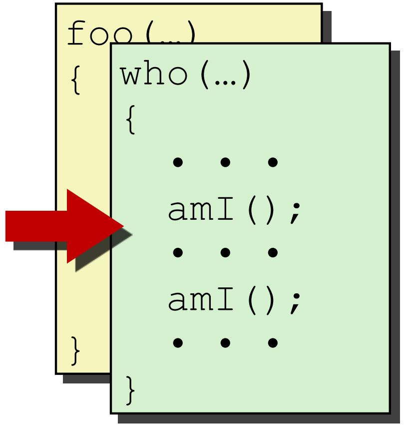




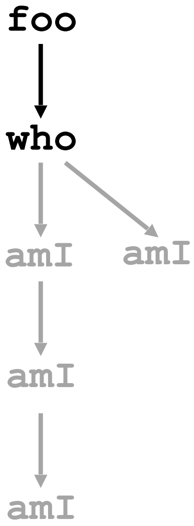
Example


foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
foo
who
amI
rbp
rsp

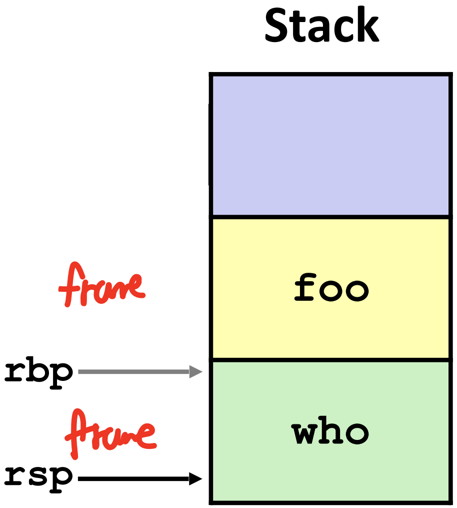
Example



foo(…)
{
}
who(…)
{
• • •
amI();
• • •
amI();
• • •
}



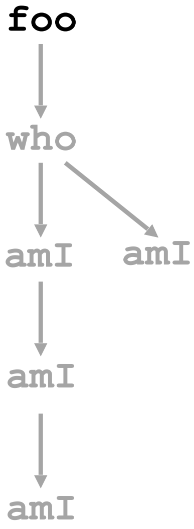
foo
who
amI
amI
amI
amI

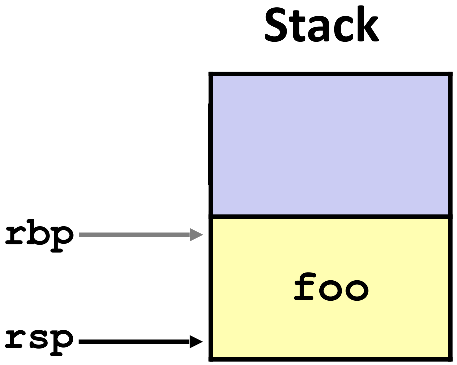


Stack
frame
rbp
frame
rsp
foo
who

Example

```
foo(…)
{
   •
   •
   who();
   •
   •
}
```





Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - Stack discipline follows call / return pattern

    by mistake
    - If P calls Q, then Q returns before P
    - Last-In, First-Out

- Also works for mutual recursion
  - P calls Q; Q calls P

Functions Summary

- Important Points
  - Stack is the right data structure for function call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result returned in **`rax`**

rbx
rbp 12
rsp 13
14