Machine-Level Programming: Procedures

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Today

- Recursion
- Recursive procedures

Recursion

- **Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration)**

- **Most computer programming languages support recursion by allowing a function to call itself**
  - Some functional programming languages do not define any looping constructs but rely solely on recursion to repeatedly call code

- **Recursive techniques can often present simple and elegant solutions to problems**

Recursion

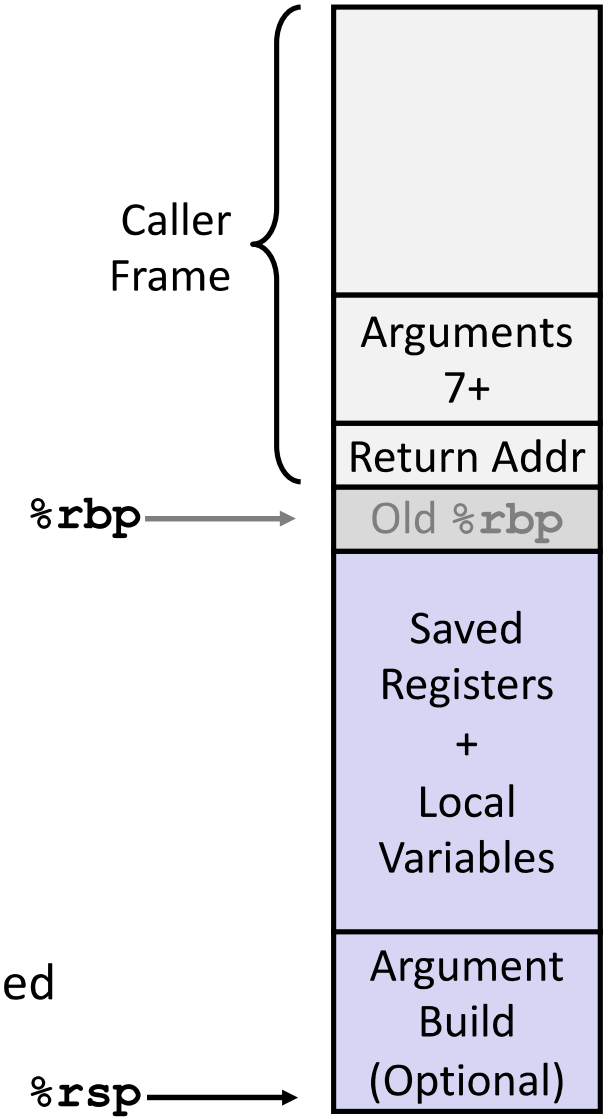
- **A recursive function definition has one or more *base cases*, and one or more *recursive cases***

- **The base case**
  - A small problem that we know how to solve and is the case that causes the recursion to end
  - The function produces a result trivially (without recurring)

- **The recursive case**
  - The more general case of the problem we're trying to solve
  - The function recurs (calls itself)
  - The recursive call must always be to a case that makes progress toward a base case

Stack support

- **The stack conventions described in the previous lectures allow procedures to call themselves recursively**
- **Each call has its own private space on the stack**
  - The local variables of the multiple calls do not interfere with one another
- **The stack discipline naturally provides the proper policy for:**
  - Allocating local storage when the procedure is called
  - Deallocating it when it returns



Call chain

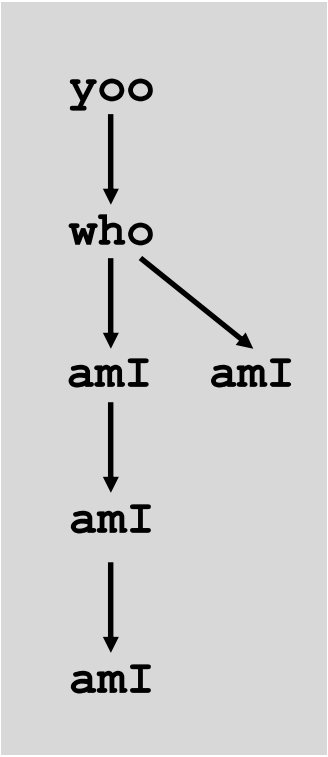
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

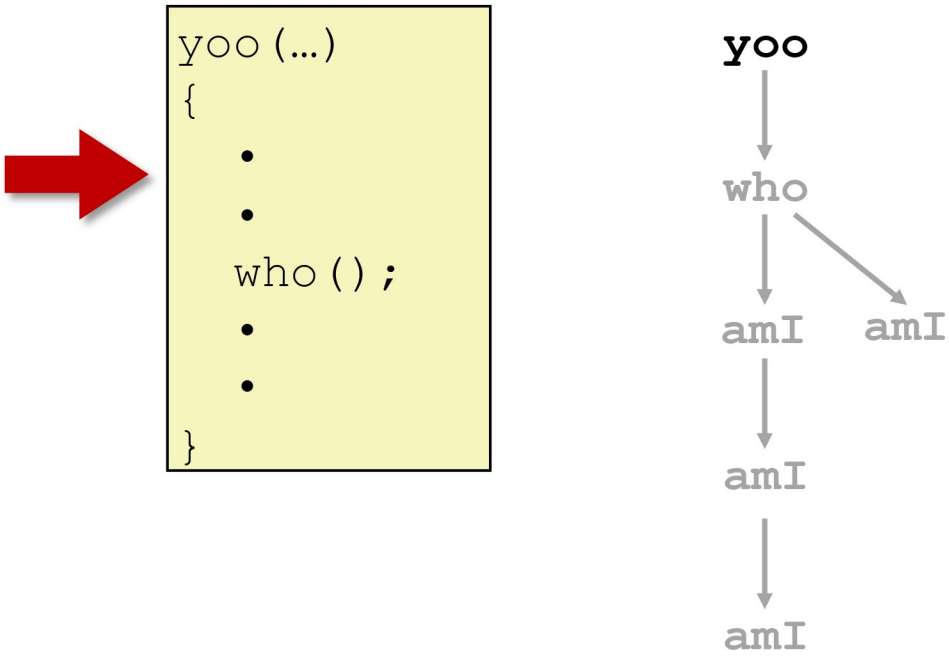
```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Procedure **amI()** is recursive

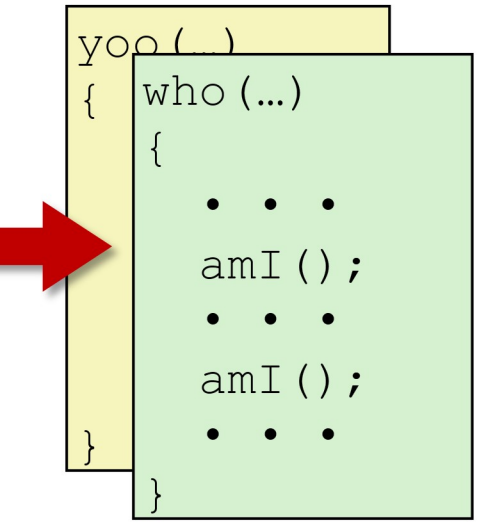
Call chain example

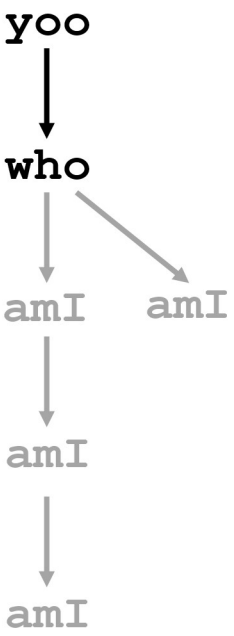


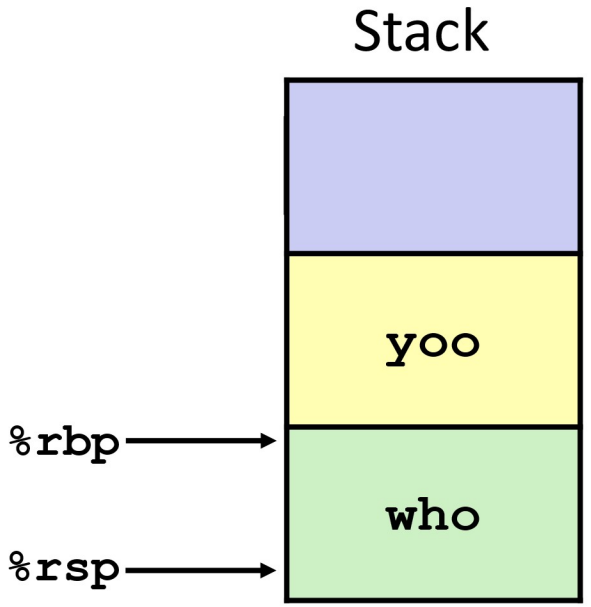
Call chain example



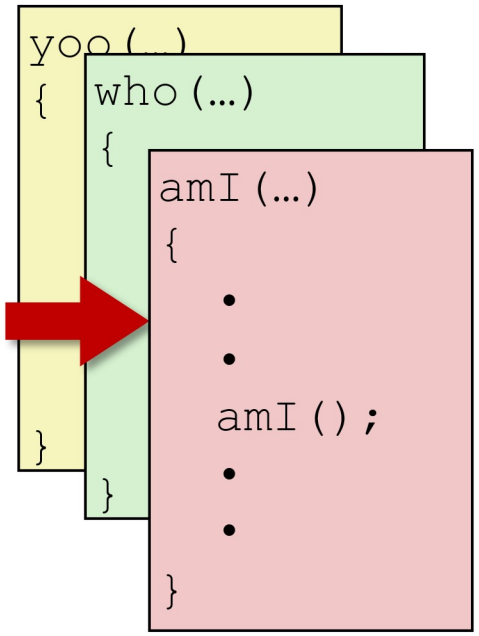
Call chain example

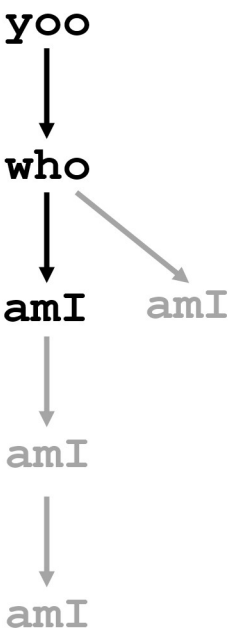


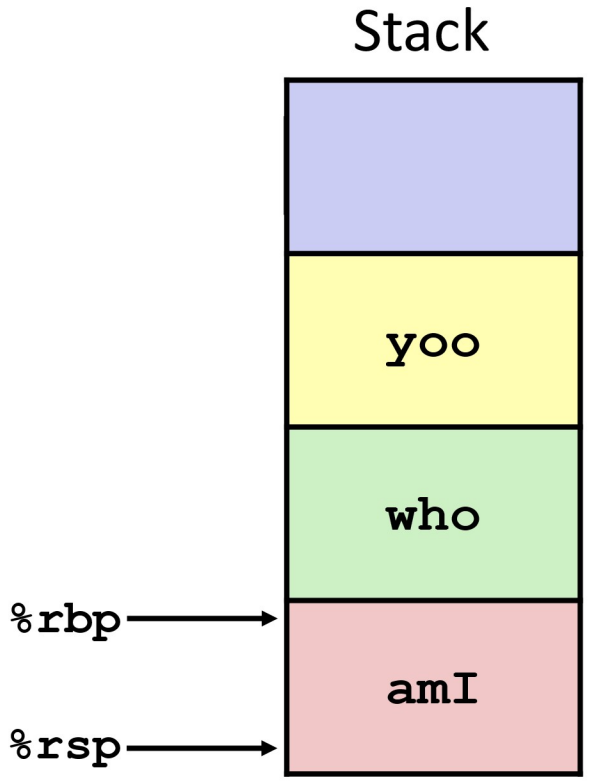




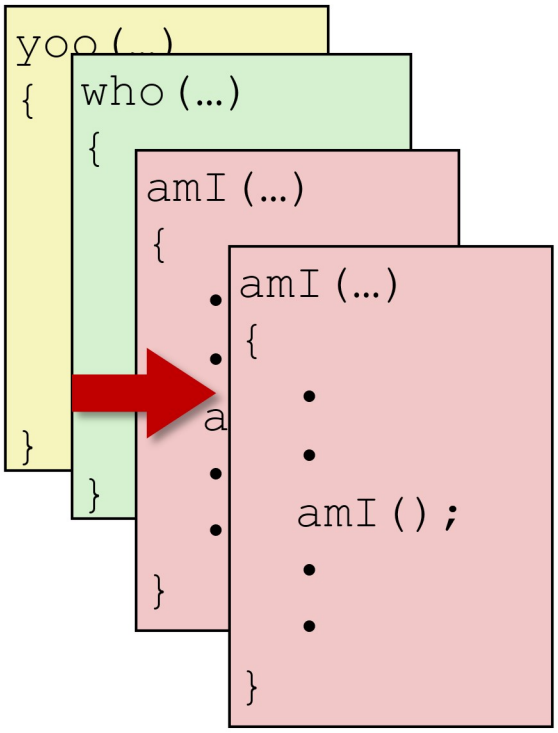
Call chain example

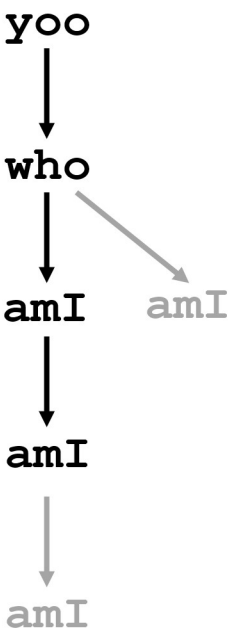


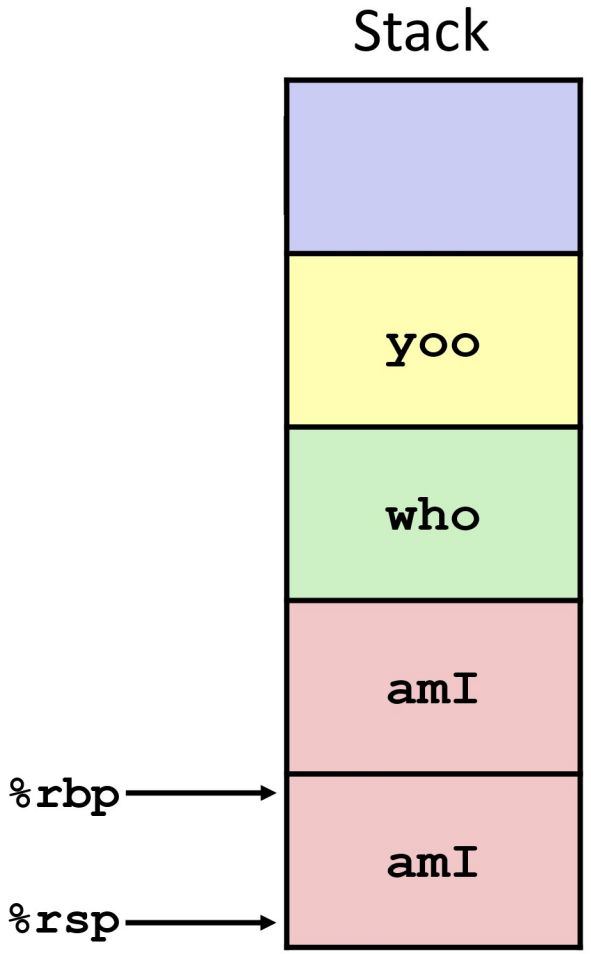




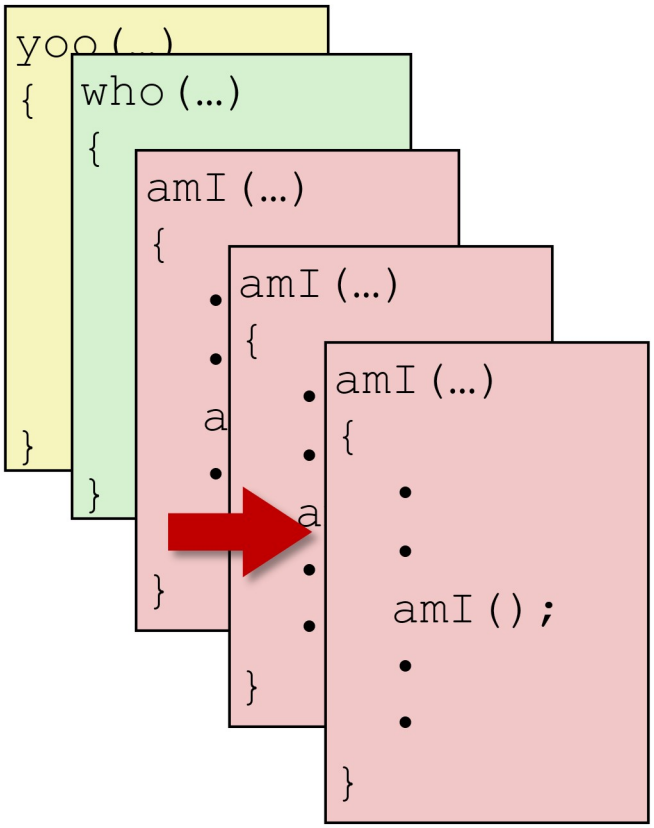
Call chain example

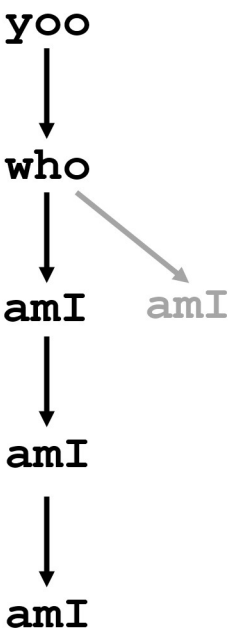


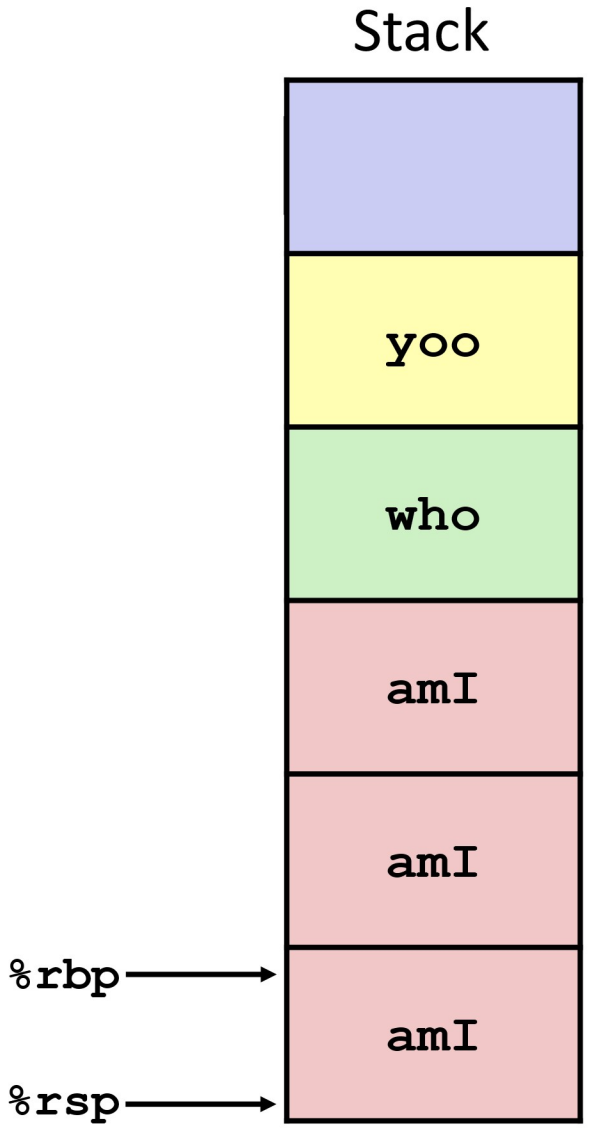




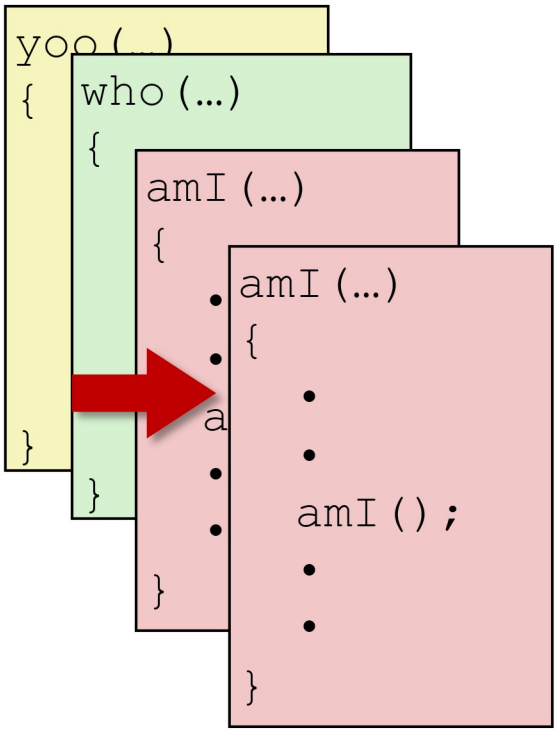
Call chain example

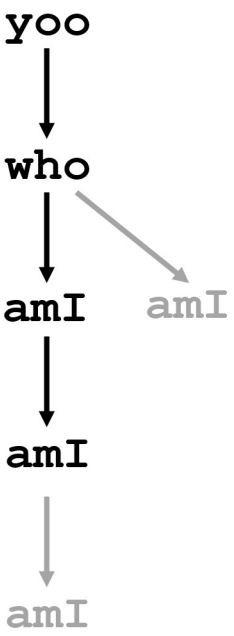


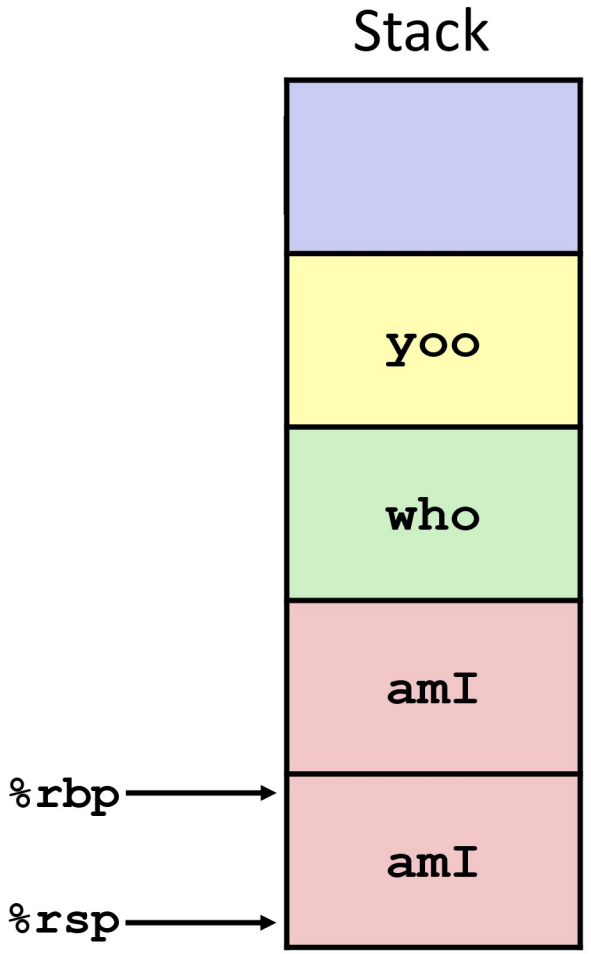




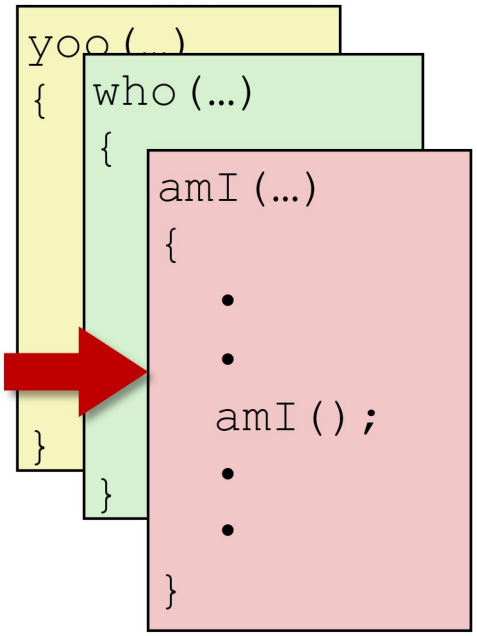
Call chain example

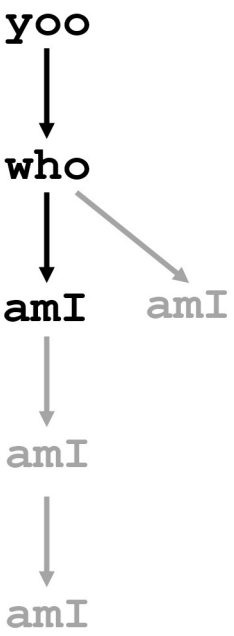


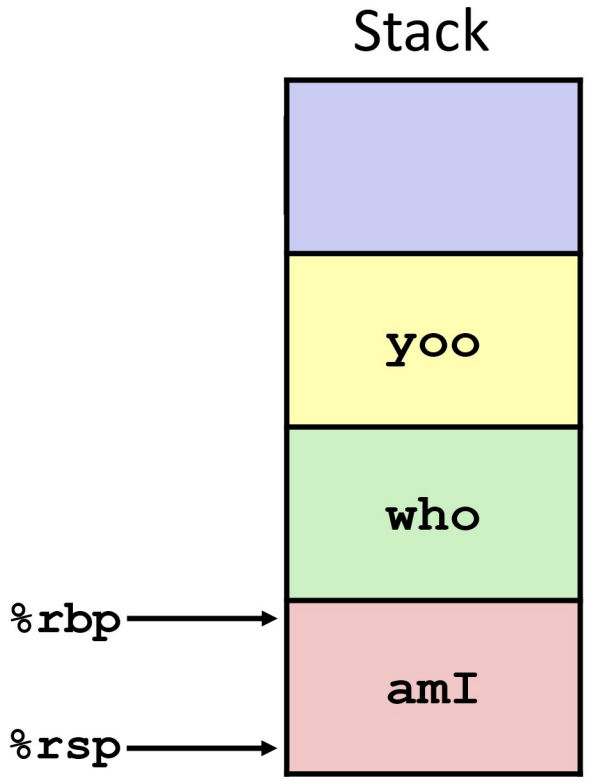




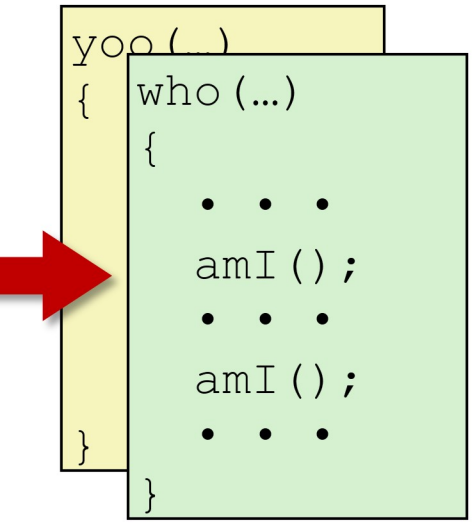
Call chain example

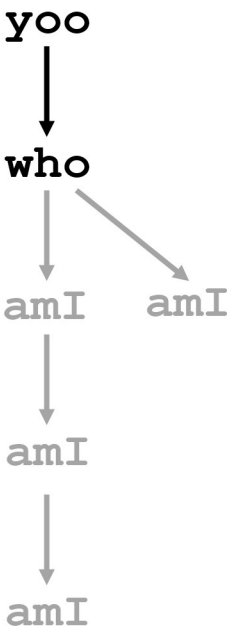


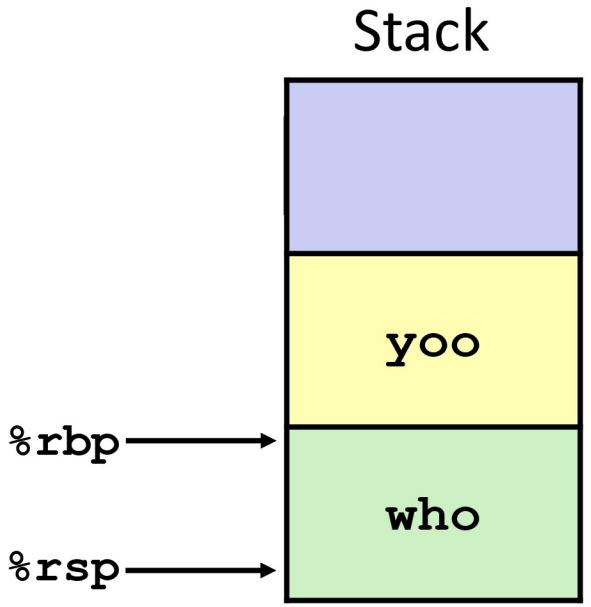




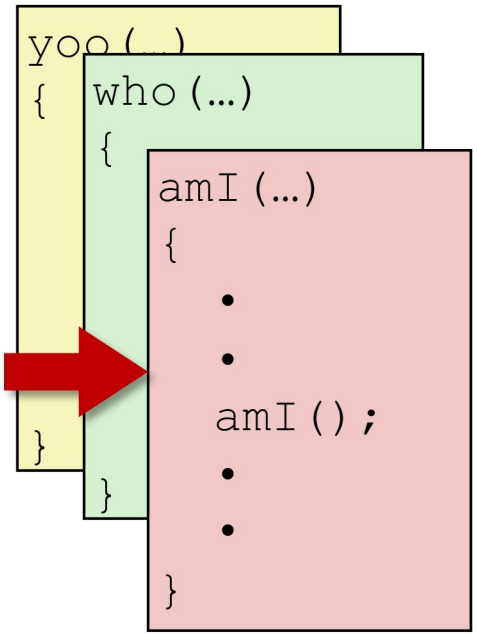
Call chain example

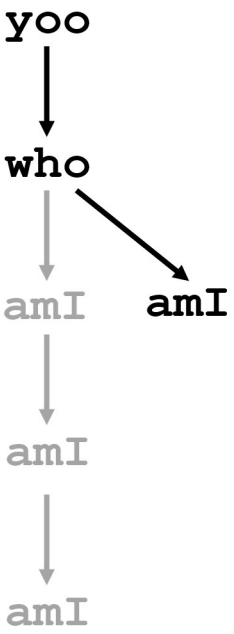


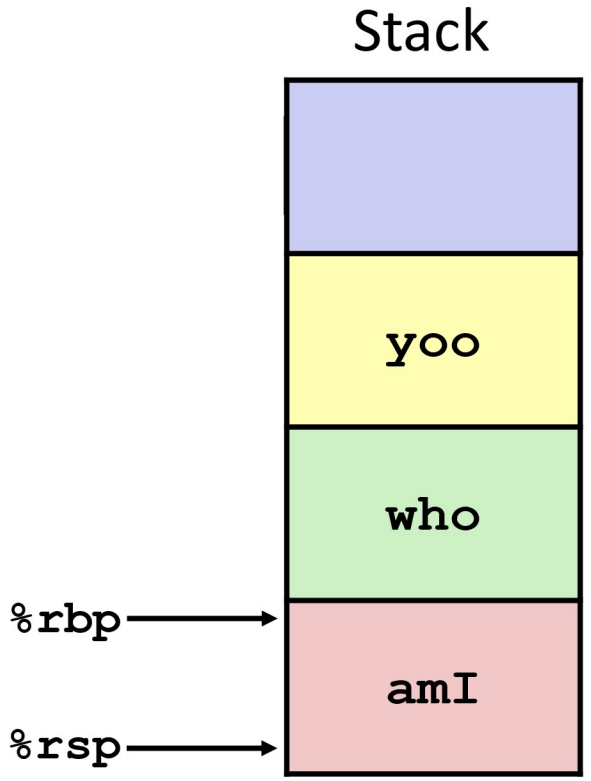




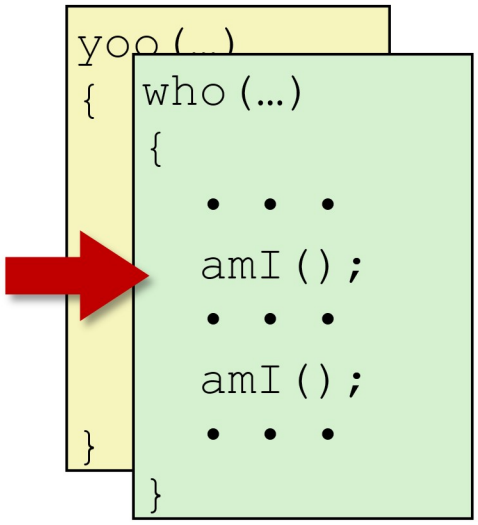
Call chain example

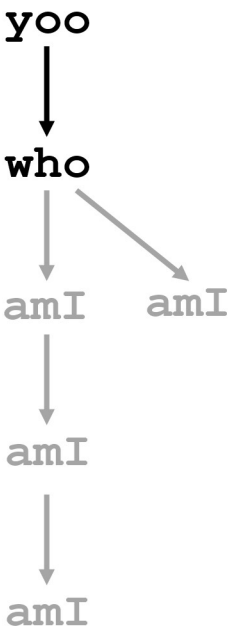


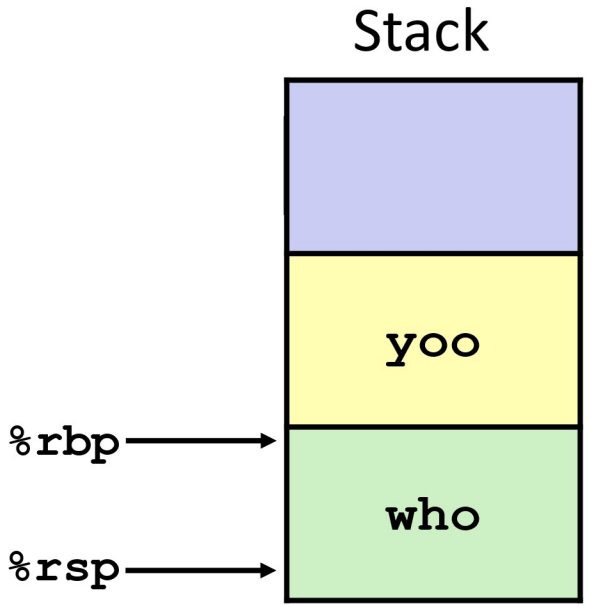




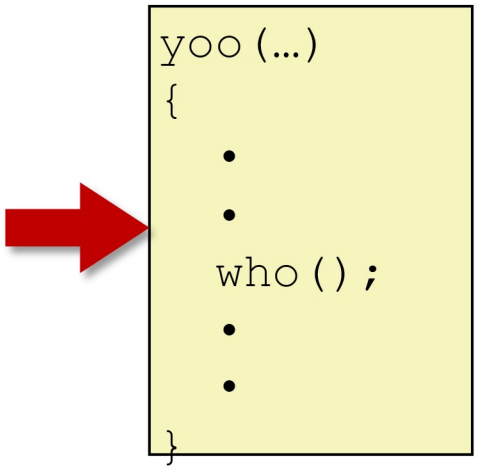
Call chain example

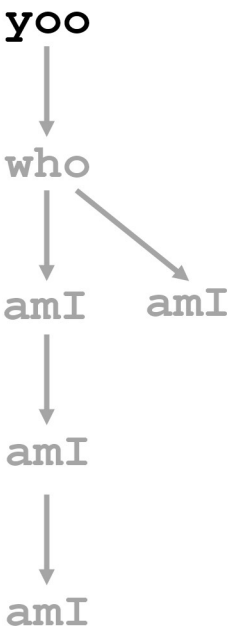


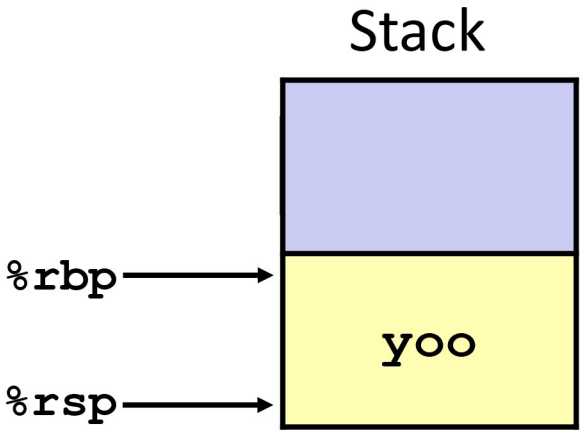




Call chain example







Today

- Recursion
- **Recursive procedures**

Recursive procedures

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Recursive procedures: Base case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

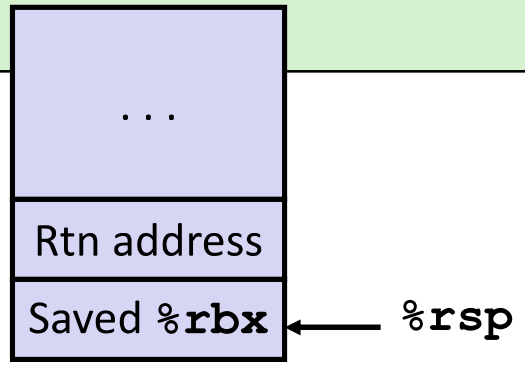
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive procedures: Register saving

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive procedures: Call setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rdi	x >> 1	Recursive argument
%rbx	x & 1	Callee-saved

Recursive procedures: Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rbx	**x & 1**	Callee-saved
%rax	Recursive call return value	

Recursive procedures: Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

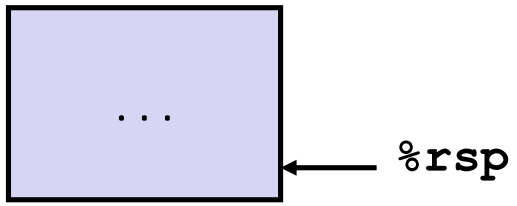
Register	Use(s)	Type
%rbx	**x & 1**	Callee-saved
%rax	Return value	

Recursive procedure: Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movq    $0, %rax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andq    $1, %rbx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rax	Return value	Return value



Observations about recursion

- **Handled without special consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers and local variables
    - Saved return pointer
- **Register saving conventions prevent one function call from corrupting another's data**
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P