CS240: Programming in C

Lecture 6: Recursive Functions
C Pre-processor.

Functions: extern and static

- Functions can be used before they are declared
- static for a function means the function is local only to that file
- extern, means that the function was declared in another file or the same file but later
- Always put prototype before definition to avoid any problems

Variables

- All variables must be declared before use
- extern has the same meaning as for functions
- static the same when declared outside functions
- static declared within a function ‘has memory’, i.e is initialized only the first time the function is called
- Don’t use the same names for global and local variables

Passing Parameters

- In C, parameters are passed to functions BY VALUE
- Functions create local copies of those variables
- Modifications are not preserved outside the functions unless the function is passed references to variables
  - int swap(int[])

Recursive functions in C

- A function can call itself
  - Recursive expression of the function
  - Needs a stop condition

  Example: compute n!

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

- Why does it work?
  - Typically used to compute an *inductively* defined property

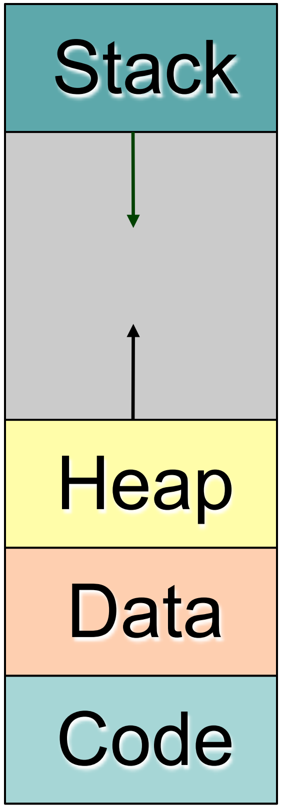
Induction

- Suppose that P is a predicate on natural numbers.
  - Suppose P(0) holds
  - And, for all i, P(i) => P(i+1)
  - Then, P(n) holds for all n.
- Let P be the “factorial property”:
  - P(0) = 1
  - P(n) = n * P(n-1)
  - If we know P(n) then we have an algorithm to compute P(n+1)
    - simply multiply n * P(n)

Operationally… it's all about the Stack

- The operating system creates a process by assigning memory and other resources
- Stack: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- Heap: dynamic memory for variables that are created with *malloc, calloc, realloc* and disposed of with *free*
- Data: initialized variables including global and static variables, un-initialized variables
- Code: the program instructions to be executed

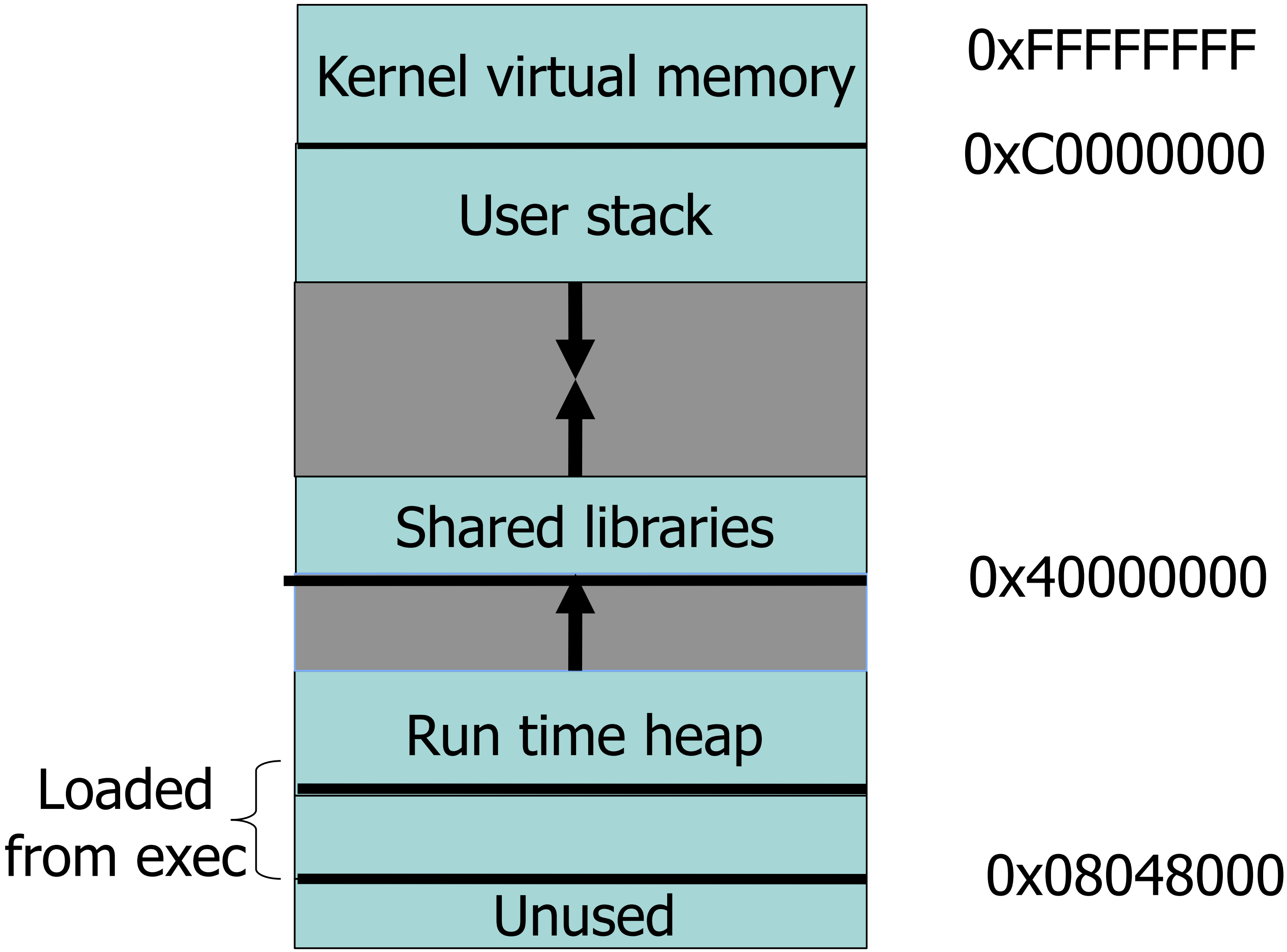
Virtual Memory



Stack

- Logically it's a LIFO structure
- Two operations: push and pop
- Grows 'down' from high-addresses to low
- Operations always happen at the top: push and pop, organized
- It is used to hold "activation frames" that represent the state of functions as they execute
  - top-most (lowest) frame corresponds to the currently executing function
- <u>It stores not only the local variables but also the address of the function that needs to be executed next</u>

Example: Linux Process Memory Layout

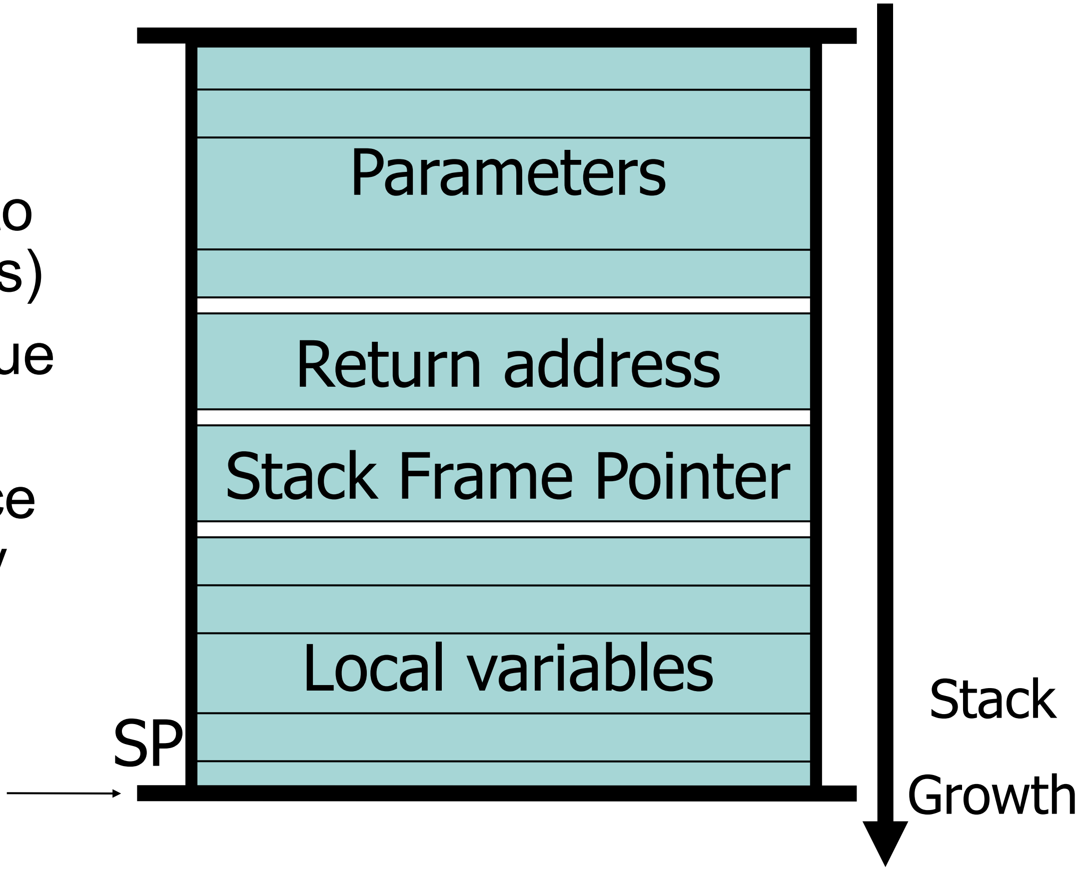


C Program execution

- PC (program counter or instruction pointer) points to next machine instruction to be executed
- Procedure call:
  - Prepare parameters
  - Save state (SP (stack pointer) and PC) and allocate on stack local variables
  - Jumps to the beginning of procedure being called
- Procedure return:
  - Recover state (SP and PC (this is return address)) from stack and adjust stack
  - Execution continues from return address

Stack frame

- Parameters for the procedure
- Save current PC onto stack (return address)
- Save current SP value onto stack
- Allocates stack space for local variables by decrementing SP by appropriate amount



Example: N!

- Observation: n! = n*(n-1)! and 1! = 1

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

Zooming in …

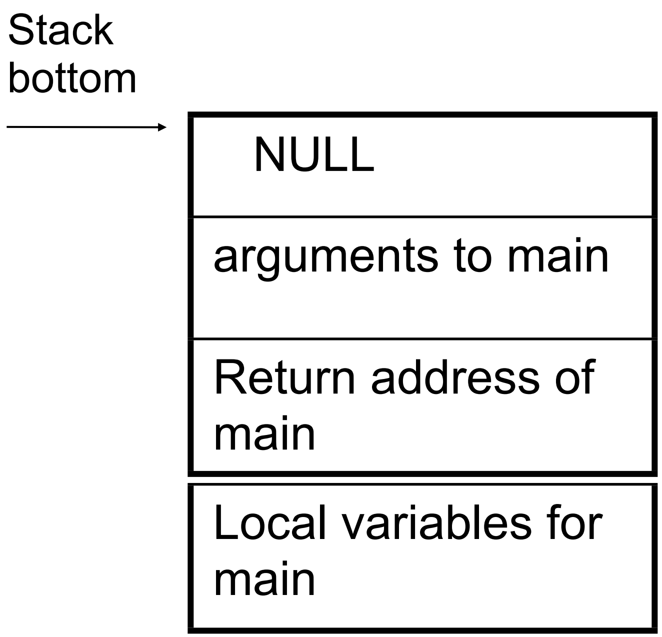
```
int factorial(int i) {
        if(i<=1) return 1;
        else return i*factorial(i-1);
}
```

factorial(3)=?

Zooming in …

```
int factorial(int i) {
        if(i<=1) return 1;
        else return i*factorial(i-1);
}
```

factorial(3)=?

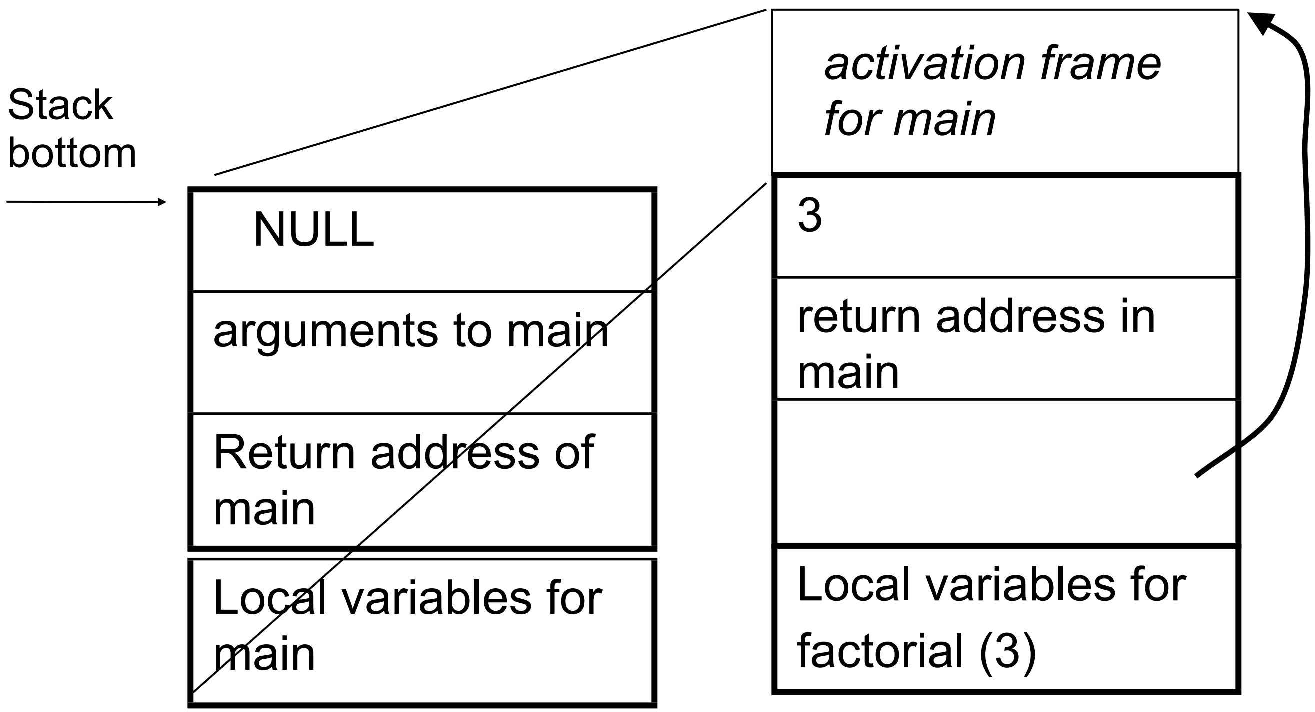


Zooming in …

```
int factorial(int i) {
        if(i<=1) return 1;
        else return i*factorial(i-1);
}
```

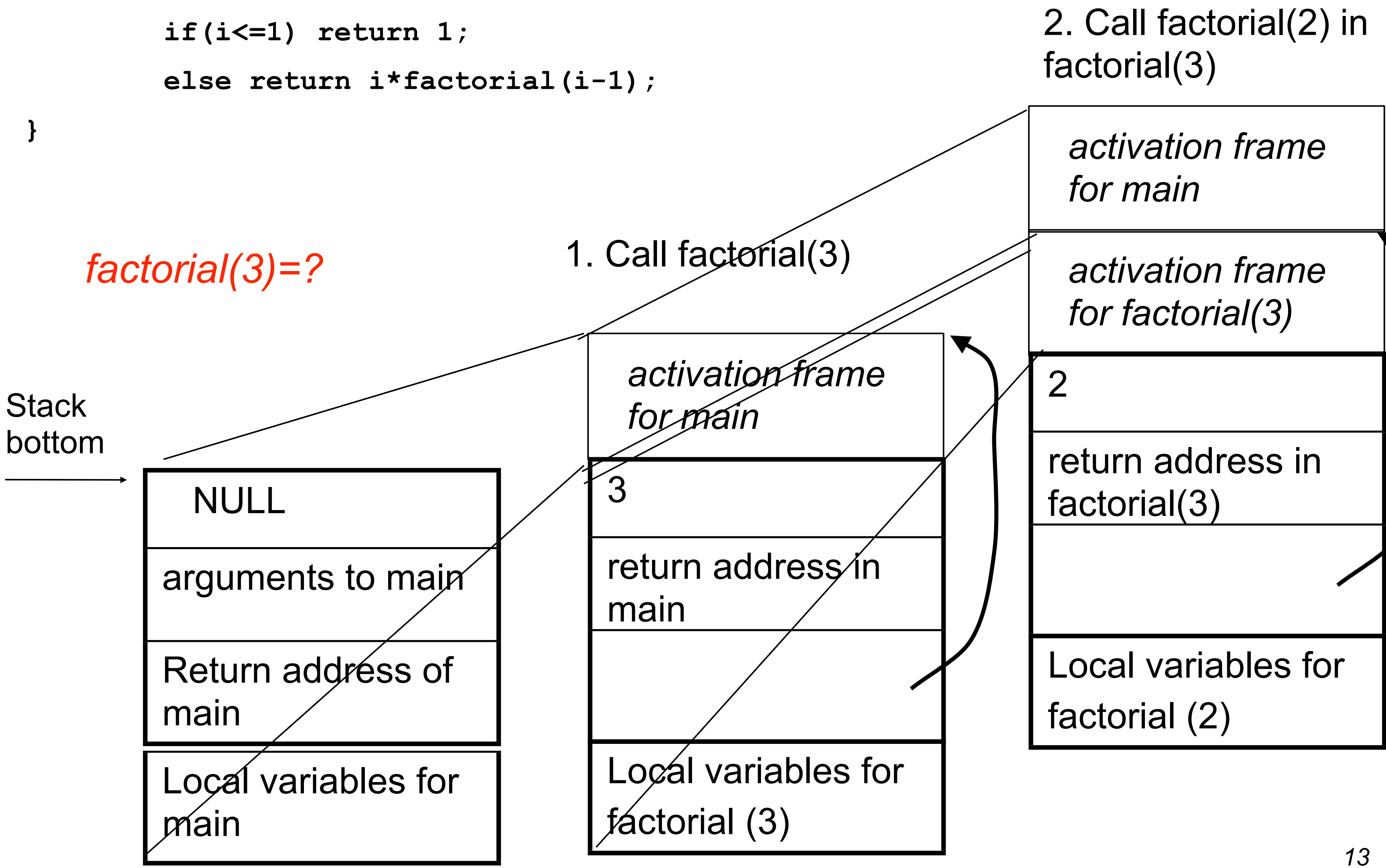
factorial(3)=?

1. Call factorial(3)

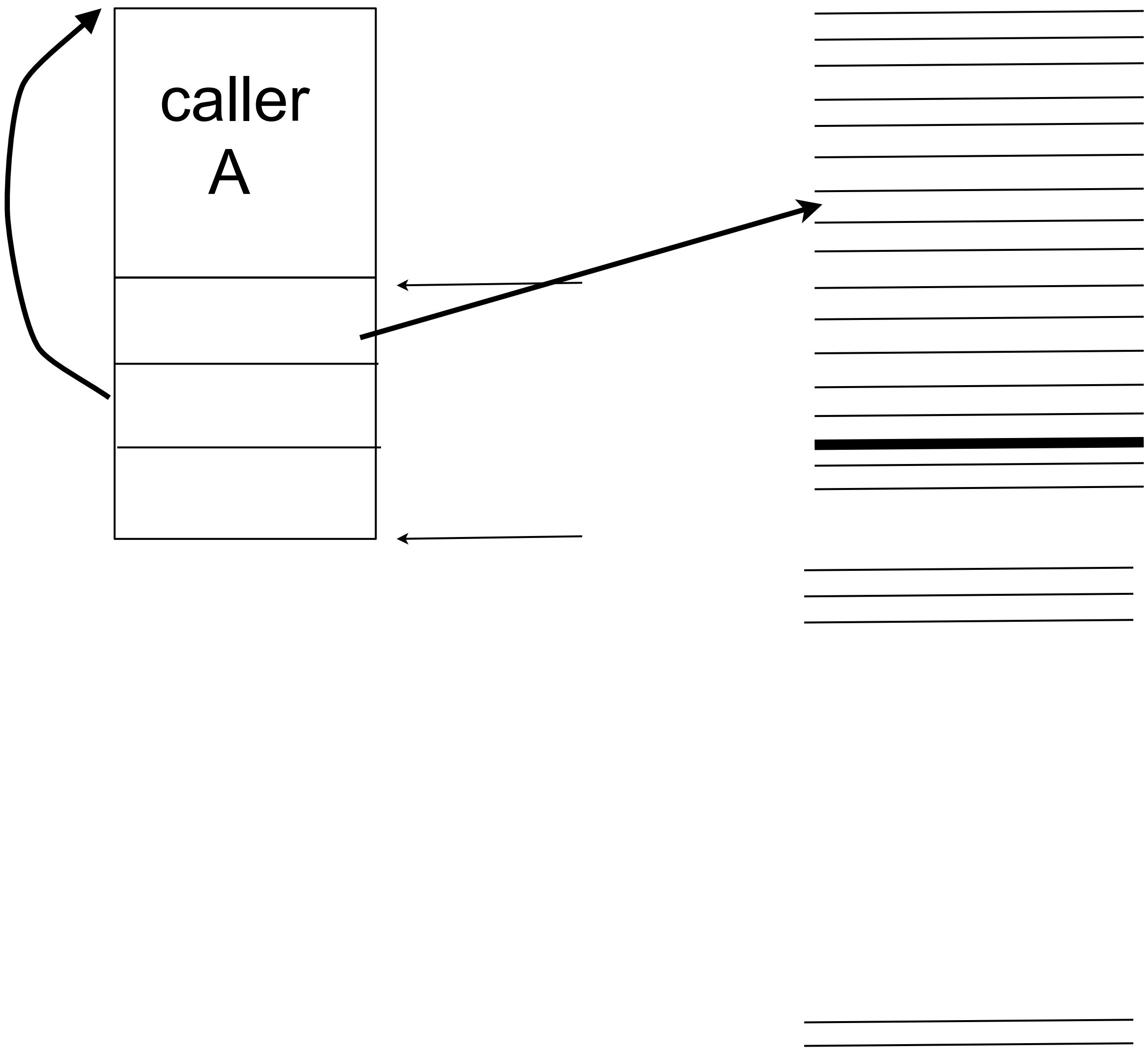


Zooming in …

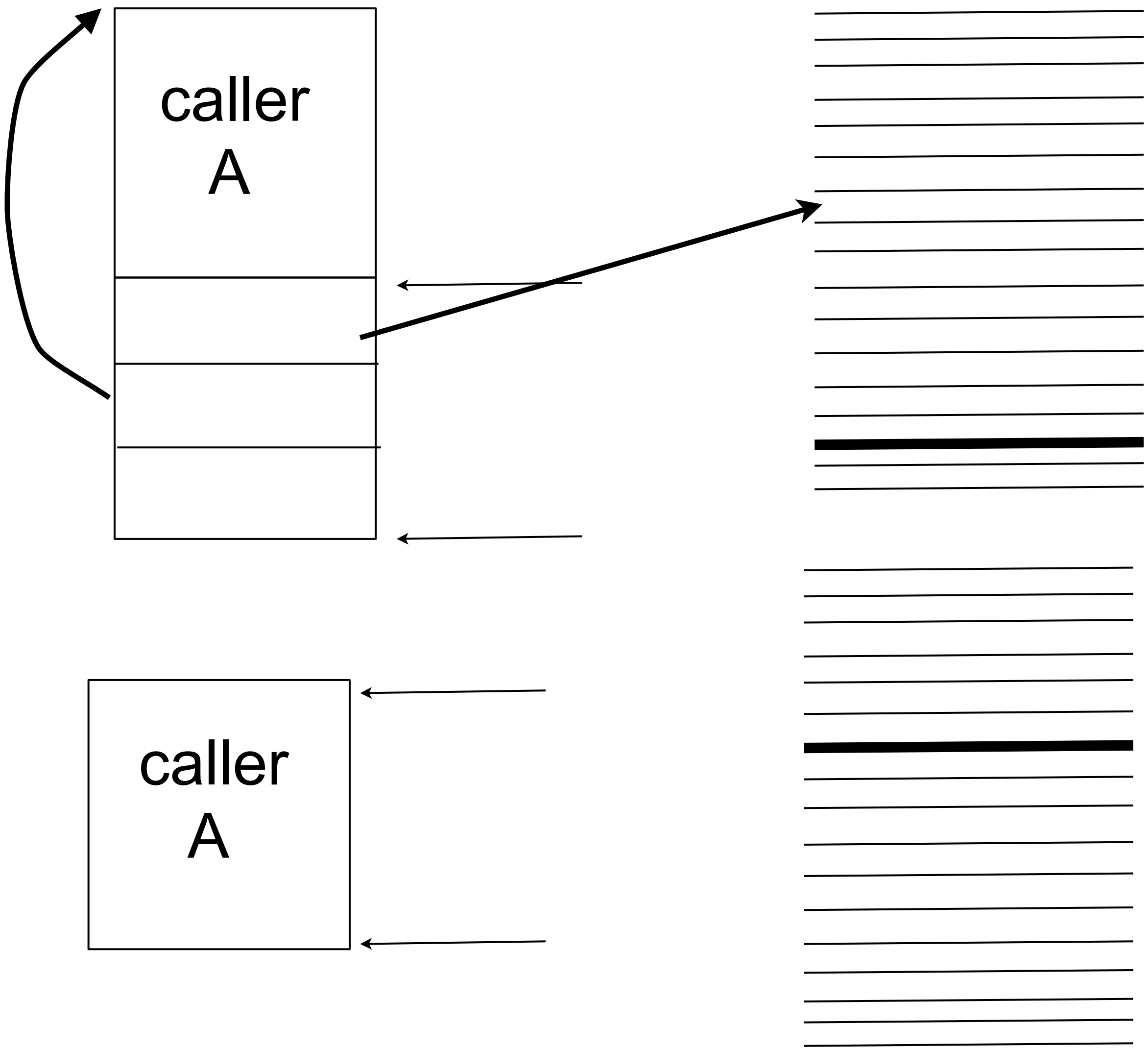
```
int factorial(int i) {
        if(i<=1) return 1;
        else return i*factorial(i-1);
}
```



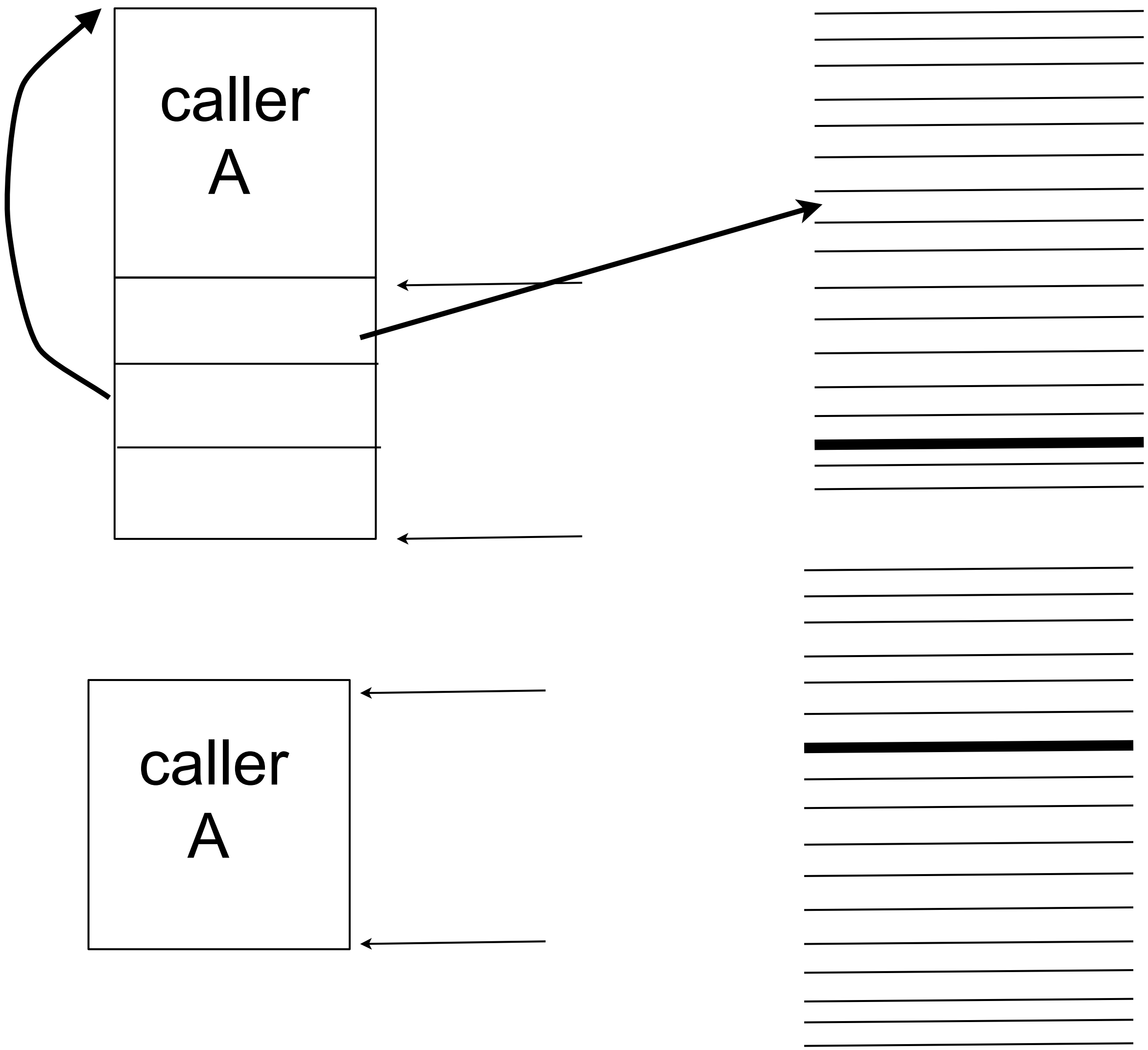
Function return



Function return



Function return



How do we pass return values back to the caller?

Typically, reserve a register for this purpose

-- EAX on x86

Tail Recursion

- Do we always need to build a new stack frame when we make a recursive call?

```
int factorial(int i, int acc) {
        if(i<=1) return acc;
        else return factorial(i-1,i*acc);
}
factorial(10,1)
```

- Notice that nothing interesting happens after the recursive call returns
  - Control is immediately transferred to the caller's caller
  - The sequence of recursive calls behaves just like a loop
  - No need to build up stack since there is no "context" or history that's preserved across calls
- C does not provide support for tail recursion
  - Recursion is not the same as looping

C Pre-processor

- Additional step before compilation
- Provides two operations
  - include
  - define

#include

- "" starts searching at source program location (within same directory);
- <> follows implementation dependent
- rules; e.g., /usr/include and -I option in gcc specified at compilation time
- included file is usually a header (.h) file, but can also be a .c file or any other file

```
#include "filename"
#include <filename>
```

Example

- You have implemented a program package with a set of functions for other programmers to call
- You distribute the implementation of your code as a library
- You distribute the interface of your code as a header file for users of your code to #include, like <stdio.h> for the standard I/O library of the libc.a C library
- The .h file contains, say, prototypes of functions that the users will call, and external variables that the users can set to control your program's behavior.

Macro substitution

- scope is from occurrence of #define to corresponding #undef, another #define of the same name, or end of file
- simple textual substitution, NO LANGUAGE AWARENESS

```
#define name replacement-text
#undef name
```

Examples

- #define STEP 10
- #define forever for (;;)
- #define max(A, B) ((A) > (B) ? (A) : (B))

Parentheses around arguments ensures correct order of evaluation under substitution

What's the essential difference between a macro and a function?

Lazy vs. strict evaluation

Consider: #define f(a,b) if (a = 0) b else 0
f 1 10/0

When #defines go wrong

- What's wrong with

```
#define square(x) x * x
```

- Recursive macros?

```
#define f(x) f((x)-1) * 2)
```

Conditional pre-processing

- #ifdef
- #ifndef
- #else
- #elif
- #endif

Applications of #ifdef: portability

```
#ifdef SYSV
#define HDR "sysv.h"
#elif defined(BSD)
#define HDR "bsd.h"
#elif defined(MSDOS)
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

Application of #idndef: include files

- To include a include file only once

```
#idndef _MY_INCLUDE_FILE_
#define _MY_INCLUDE_FILE_
 header file

#endif /* _MY_INCLUDE_FILE_ */
```

Application of #ifndef: include files

- To include a include file only once

```
#ifndef _MY_INCLUDE_FILE_
#define _MY_INCLUDE_FILE_
 header file

#endif /* _MY_INCLUDE_FILE_ */
```

Application of #ifdef: Print debug information

```
#ifdef DEBUG
#define DPRINTF(args) printf args
#else
#define DPRINTF(args)
#endif
```

- Specify how you want to macro to expand by specifying the DEBUG variable at compilation time in the Makefile
- gcc -D option

Readings for This Lecture

K&R Chapter 4