Run-Time Environments

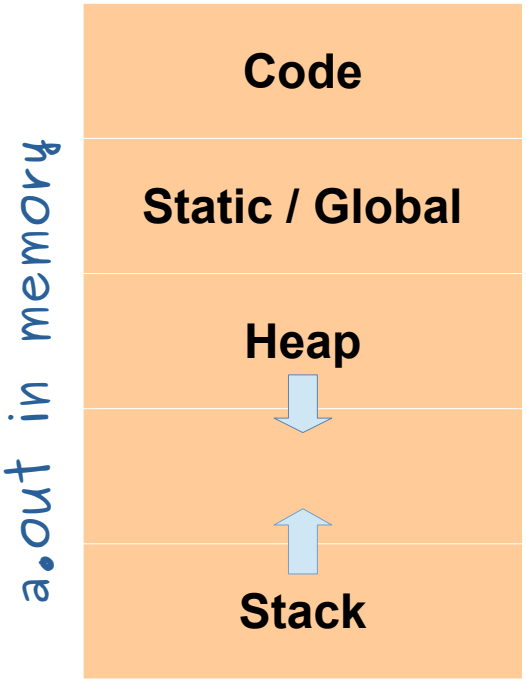
Rupesh Nasre.

CS3300 Compiler Design
IIT Madras
July 2024

Static versus Dynamic

- Time: compilation versus execution, preprocessor versus compilation
- Compilation: gcc versus jit
- Optimization: without and with input
- Analysis: without and with environment
- Type:
  - strongly typed versus scripting languages
  - inheritance and virtual functions
- Linking: .a versus .so
- Scoping

Memory Organization



Default stack size is limited (8 MB).

- Code is often rx and not w.
- Static / Global section is rw.
- const data is often not stored in code section.
  - We can fool the compiler via pointers and type-casts.
  - But a run-time environment may decide to mark the page read-only (r--).
- static in C is an abstraction over globals.
  - Hardware does not enforce it.

Scope and Lifetime

Variable	Scope	Lifetime
auto (local variables)	Function / Block	Function / Block
global	Global	100%
static local	Function / Block	100%
static global	File	100%
Dynamically allocated (malloc)	Global (via pointers)	Until deallocated (free) or 100%

Heap

- Dynamic memory allocation
  - malloc, calloc, realloc, free
  - **malloc** does not initialize memory. In Linux, it may not even allocate memory (over-provisioning).
  - **calloc** initializes memory to 0.
  - **free** does not need size. Double free has undefined behavior. Freeing a null pointer has no effect.
  - **realloc** may expand the existing memory or allocate new memory and copy the data.
  - Not recommended: realloc can be used as malloc as well as free.

Memory Manager

- Heap uses a memory manager.
  - dlmalloc (general purpose allocator, earlier Linuses)
  - ptmalloc2 (glibc, thread-friendly, per thread heap)
  - jemalloc (freebsd and firefox)
  - tcmalloc (google)
  - libumem (solaris)
  - …
- The implementations interface between user program and OS.
  - Gets arenas from OS and maintains those for a.out.

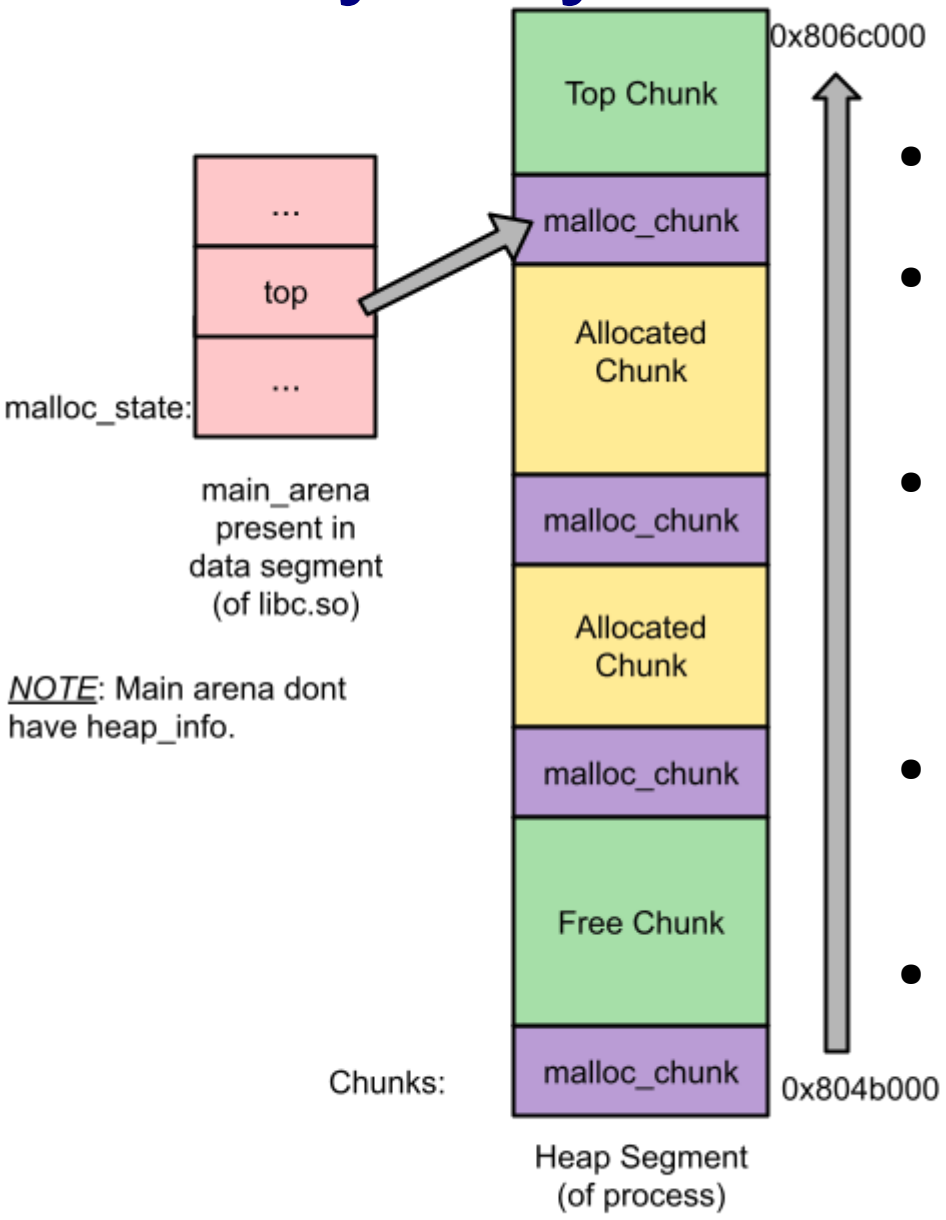
Source and code

malloc

- For small chunks, malloc uses brk / sbrk syscalls.
  - Linux uses sbrk as a library function, which calls brk as a syscall.
  - sbrk can be used to query the current break value, which indicates the end of the data segment.
- For larger chunks (> 128 KB), malloc uses mmap.
  - More communication with OS.
  - Hence, costlier.
- .bss section may get mapped to uninitialized block.
  - Useful to not allocate page, since data is not written yet.
  - On Linux, the physical pages are initialized to zero, but C does not guarantee it.

Source and code

Memory Layout with malloc and free



- mallocs lead to a linked list.
- Head pointer to the allocated list is in libc.so.
- Metadata about allocated chunk is stored prior to the allocated region.
- No two free chunks are adjacent.
- Different sized free chunks are maintained in different bins (small, large, etc.)

Main Arena

Source and code

Allocation Strategies

- malloc: where to allocate this memory when we have multiple options?

- **First-fit**: find the first free (from starting) block to satisfy the request.
- **Next-fit**: similar to first fit, but start from the end of the previously allocated chunk.
- **Best-fit**: find the smallest free chunk satisfying the request, and allocate to the left.
- **Worst-fit**: find the largest free chunk satisfying the request, and allocate centrally in it.

What is the use of next-fit and worst-fit?

free

- Deallocates memory created using malloc
  - Uses munmap if memory was allocated with mmap.
  - Does not need size of the allocated region. Uses metadata stored *prior to* the pointer.
- Free results in merging with adjacent free regions.
- Freeing nullptr is a no-op.
- Freeing previously freed memory may lead to undefined program behavior.
- Free creates dangling pointers.

```
#define myfree(p)  free(p), p = nullptr
```

Garbage Collection

- Due to the issues with free, some languages prefer to not permit users to use it.
  - Or users do not need to worry about freeing memory.
- A managed run-time is needed to collect *garbage*.
  - e.g., Java, Python, most functional languages
- GC based on algorithm
  - Reference Counting, Mark and Sweep
- GC based on static vs. dynamic behavior
  - Incremental, Stop the World
- Concepts applicable in other contexts.
  - fopen-fclose, init-destroy, lock-unlock, connect-disconnect, ...

Stack

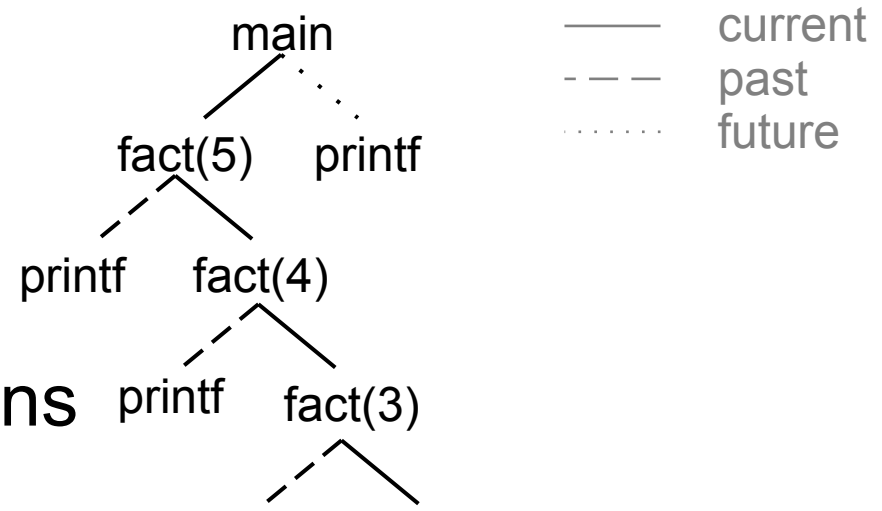
- LIFO, useful for storing function call data
  - Called as activations, recent activation on top
- Permits reuse of memory area across functions
- Permits relative addresses of variables to be the same within a function, irrespective of the call.
- Compiler needs to generate code to use a stack.
  - A VM may also use a stack to execute instructions.
- Used in exception handling

Allocations on Stack

- Local variables, temporaries, register spills, function parameters (actual), function arguments (formal), return value, return address, call environment
- Not allocated: global, static, malloc
  - Optimizations may convert heap-to-stack allocation.
- Dynamically sized arrays? Two ways:
  - Decide addresses on the fly (of the array and the latter variables).
  - Statically allocate a pointer.
- Small allocations are okay and expected.
- Large allocations are expected to be on heap.
  - Stack size needs to be updated in case needed.
  - Source: ulimit.c

Why Stack for Activations?

- The activations form a tree.
- Functions begin in preorder.
  Functions exit in postorder.
- Current sequence of activations is a path from the root.
- The DFS needs a stack.




```
int fact(int n) {
        printf("%d\n", n);
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```

An activation record is created for each function call.

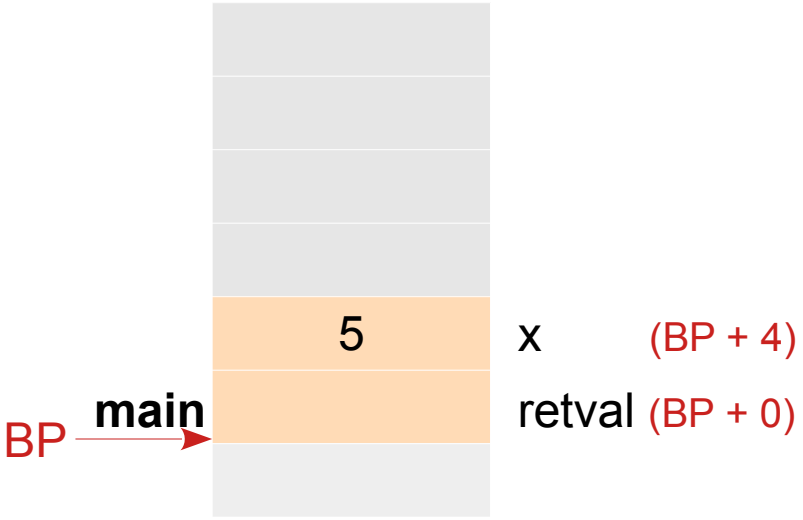
Activation Record

- Saved machine status includes return address (PC), calleX-saved registers
- Access link points to the activation record where data may be found (e.g., globals, nested procedures in static scoping)
- Control link → caller

Activation Record

Actual parameters
Return values
Control link
Access link
Saved machine status
Local data
Temporaries

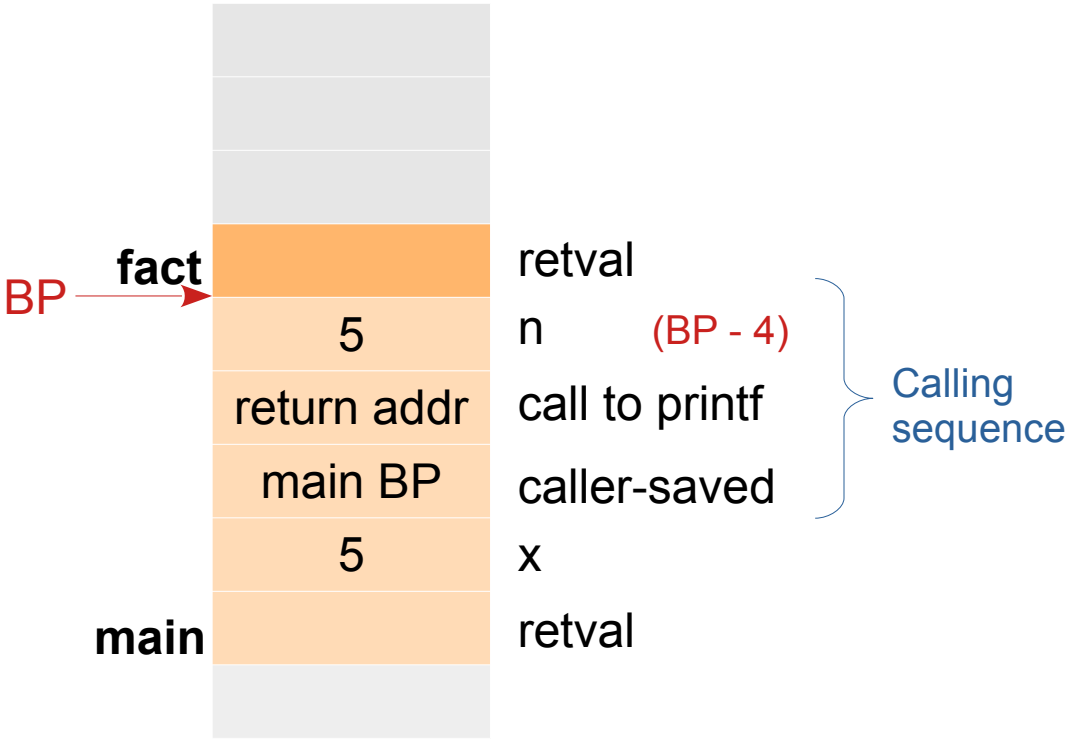
```
int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```



Activation Record

Actual parameters
Return values
Control link
Access link
Saved machine status
Local data
Temporaries

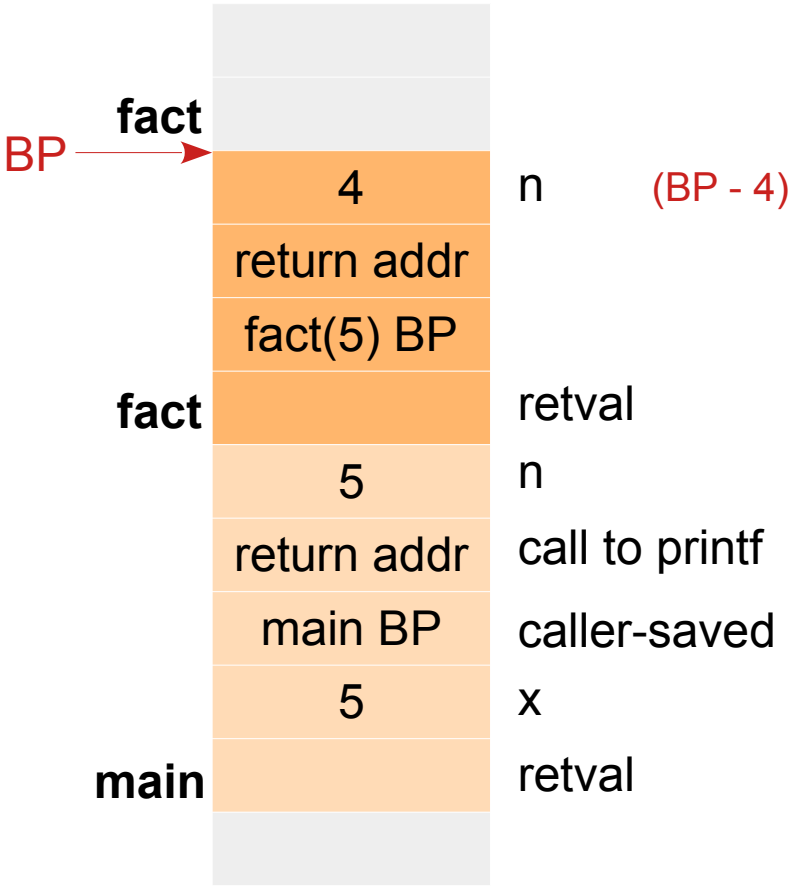
```
int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```



Activation Record

Actual parameters
Return values
Control link
Access link
Saved machine status
Local data
Temporaries

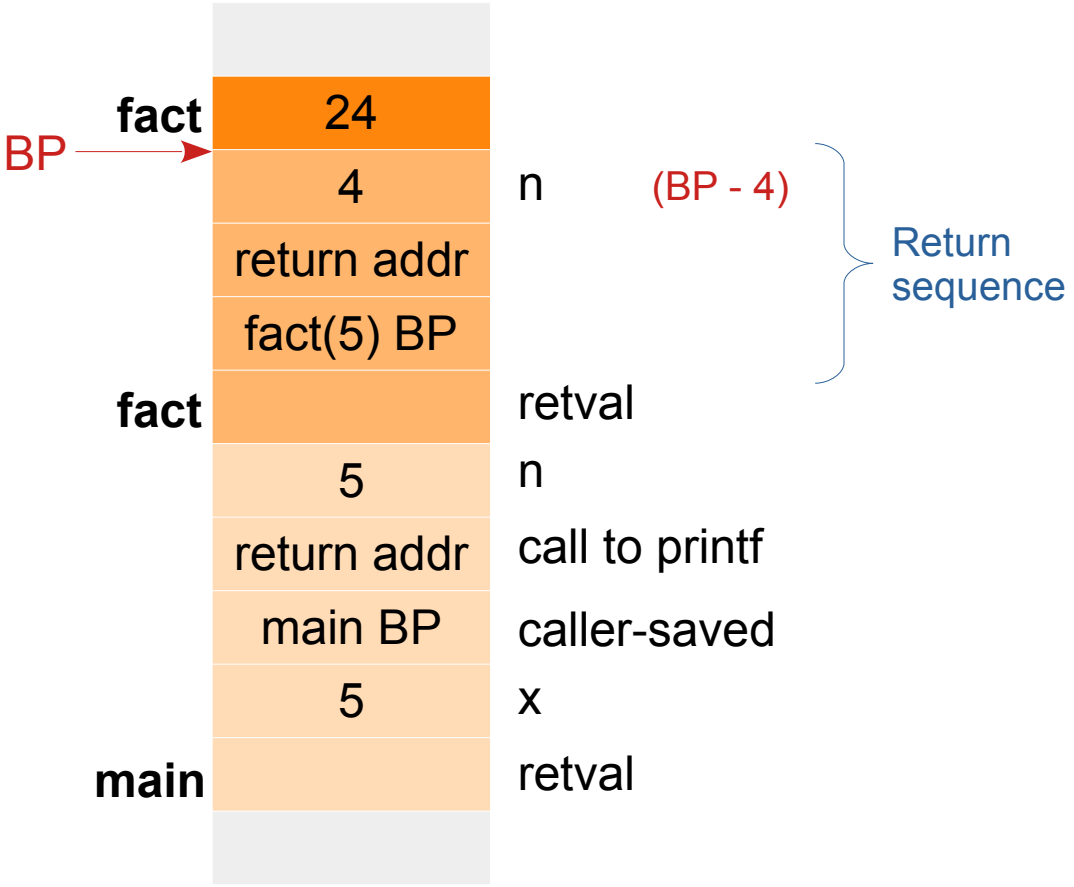
```
int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```



Activation Record

Actual parameters
Return values
Control link
Access link
Saved machine status
Local data
Temporaries

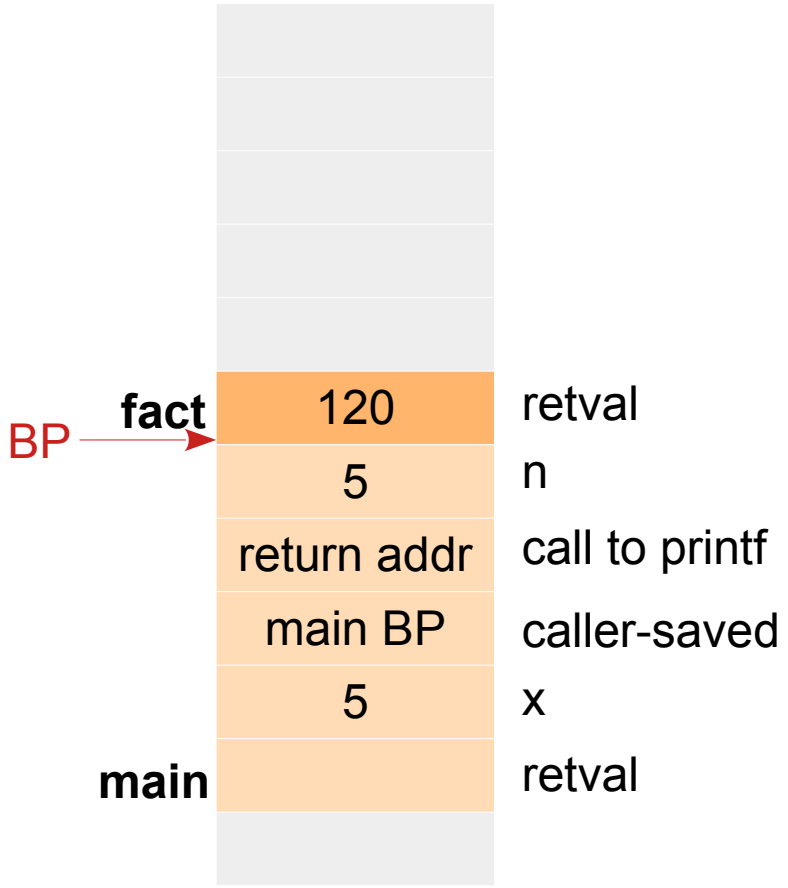
```
int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```



Activation Record

Actual parameters
Return values
Control link
Access link
Saved machine status
Local data
Temporaries

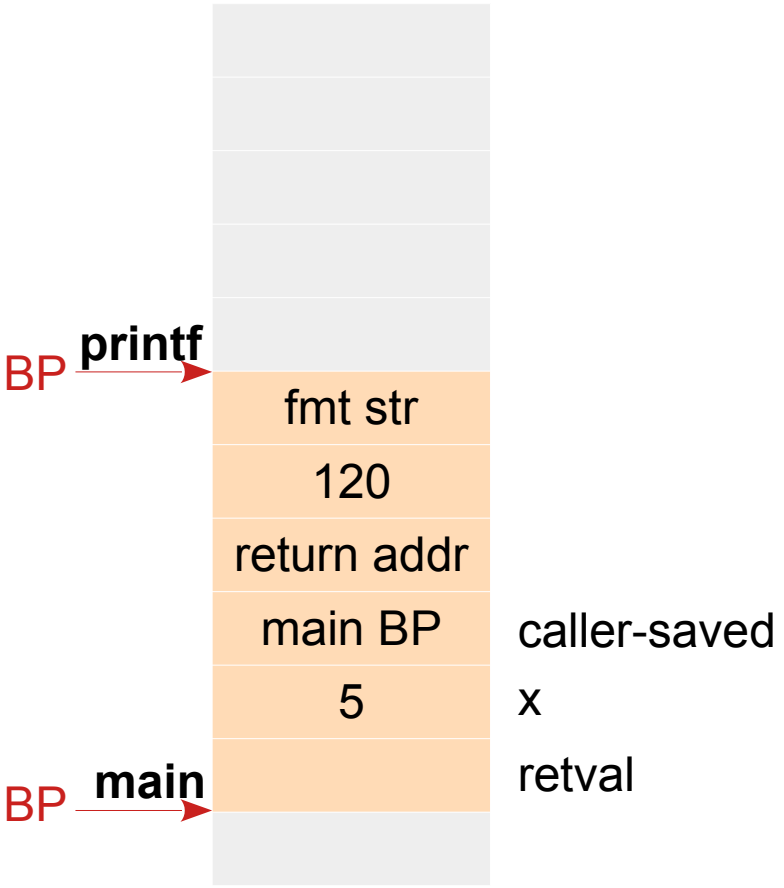
```
int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```



Activation Record

Actual parameters
Return values
Control link
Access link
Saved machine status
Local data
Temporaries

```
int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() {
        int x = 5;
        printf("%d\n", fact(x));
}
```



Register Saving

- Calling Sequence and Return Sequence
- Caller-Saved
  - Callers know what registers they want to be restored.
  - But they don't know what all callee would require.
- Callee-Saved
  - Callee knows which registers are needed.
  - Independent of the multiple callers.
  - But, it doesn't know which of the callers are using those registers.
- While the compiler may have access to both caller and callee codes, the code generation is often function-by-function.
  - Inter-procedural optimizations with -O3

Stack Depth

- CS1111: Write a program to recursively search in an unsorted array.
- Stack depth affects whether your program crashes on large inputs.
  - Your and company's credibility is at stake.
  - How can you help?
- Algorithmic considerations may affect.
  - Source: search.c // borrowed from Dr. Meghana
  - Use recursion carefully.
  - Recall PDS Trees: Tree-height not only affects execution time, it also affects stack requirement.

Frontend

Character stream

Lexical Analyzer

Token stream

Syntax Analyzer

Syntax tree

Semantic Analyzer

Syntax tree

Intermediate Code Generator

Intermediate representation

Machine-Independent Code Optimizer

Intermediate representation

Code Generator

Target machine code

Machine-Dependent Code Optimizer

Target machine code

Backend

Symbol Table

Going Forward

- Advanced Compiler Design
- Program Analysis
- Program Verification
- Programs and Proofs
- Research on various backends
  - analysis, optimization, and code generation
  - LLVM and MLIR
  - Newer architectures
- Acknowledgments