CS31: Introduction to Computer Systems

Week 6, Class 2
Functions and the Stack
02/29/24

Dr. Sukrit Venkatagiri
Swarthmore College
Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Maze Lab</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td><strong>Spring Break</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

- **C** programming language
- **x86 Assembly** instruction set architecture
- **Binary** logic / bits
- **CPU / memory** logic / bits
- **logic gates, circuits** voltage
Reading Quiz
The portion of the stack allocated for a single function is known as a...

A. function block
B. function segment
C. stack pointer
D. stack frame
The stack frame for the currently-executing function is bounded by the stack pointer and the ___.

A. frame pointer

B. bounds pointer

C. function pointer

D. stack pointer #2
Using a stack to record function calls allows a language to easily support...

A. iteration  
B. recursion  
C. pointers  
D. gotos
Overview

• Stack data structure, applied to memory

• Behavior of function calls

• Storage of function data, at assembly level
“a” Stack

• A stack is a basic data structure
  • Last in, first out behavior (LIFO)... just like a stack of papers
  • Two operations
    • Push (add item to top of stack)
    • Pop (remove item from top of stack)
“the” Stack

- Apply stack data structure to memory
  - Store local (automatic) variables
  - Maintain state for functions (e.g., where to return)

- Organized into units called *frames*
  - One frame represents all of the information for one function
  - Sometimes called *activation records*
Memory Model

• Starts at the highest memory addresses, grows into lower addresses
Stack Frames

• As functions get called, new frames added to stack

• Example: Lab 4
  • main calls get_values()
  • get_values calls read_float()
  • read_float calls I/O library
Stack Frames

• As functions return, frames removed from stack

• Example: Lab 4
  • I/O library returns to read_float
  • read_float returns to get_values
  • get_values returns to main

All of this stack growing/shrinking happens automatically (from the programmer’s perspective)
What is responsible for creating and removing stack frames? Why?

A. The user

B. The compiler

C. C library code

D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.
Stack Frame Contents

• What needs to be stored in a stack frame?
  Alternatively: **What must a function know / access?**

• Local variables
Local Variables

If the programmer says:

```
int x = 0;
```

Where should x be stored?

(Recall basic stack data structure)

Which memory address is that?

```

function 1

function 2

main

X goes here

0xFFFFFFFF
```

0xFFFFFFFF
How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location

B. The CPU stores the location of the current stack frame

C. The operating system keeps track of the top of the stack

D. The compiler knows / determines where the local data for each function will be as it generates code

E. The address is determined some other way
Program Characteristics

• Compile time (static)
  • Information that is known by analyzing your program
  • Independent of the machine and inputs

• Run time (dynamic)
  • Information that isn’t known until program is running
  • Depends on machine characteristics and user input
The Compiler Can...

- Perform type checking

- Determine how much space you need on the stack to store local variables

- Insert assembly instructions for you to set up the stack for function calls
  - Create stack frames on function call
  - Restore stack to previous state on function return
Local Variables

- Compiler can allocate N bytes on the stack by subtracting N from the “stack pointer” (moving the stack pointer “up”): %rsp
The Compiler Can’t…

• Predict user input

```c
int main(void) {
    int decision = [read user input];
    if (decision > 5) {
        funcA();
    } else {
        funcB();
    }
}
```
The Compiler Can’t…

- Predict user input.

```c
int main(void) {
    int decision = [read user input];
    if (decision > 5) {
        funcA();
    } else {
        funcB();
    }
}
```
The Compiler Can’t…

- Predict user input

- Can’t assume a function will always be at a certain address on the stack

Alternative: create stack frames relative to the current (dynamic) state of the stack
Stack Frame Location

- Where in memory is the current stack frame?
Recall: x86_64 Register Conventions

- Working memory for currently executing program
  - Temporary data ( %rax - %r15 )

- **Location of runtime stack** (%rbp, %rsp)

- Address of next instruction to execute ( %rip )

- Status of recent ALU tests ( CF, ZF, SF, OF )

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
<th>%r14</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%r9</td>
<td>%r15</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
<td></td>
</tr>
</tbody>
</table>

General purpose registers

- Current stack top
- Current stack frame
- Program Counter (PC)

Condition codes (flags)
Stack Frame Location

• Where in memory is the current stack frame?

• Maintain invariant:
  • The current function’s stack frame is always between the addresses stored in \texttt{rsp} and \texttt{rbp}

• \texttt{rsp}: stack pointer
• \texttt{rbp}: frame pointer (base pointer)
Stack Frame Location

• Compiler ensures that this invariant holds
  • We’ll see how a bit later

• This is why all local variables we’ve seen in assembly are relative to rbp or rsp!

```
main
    function 1
    function 2
```

0xFFFFFFFF
How would we implement pushing x to the top of the stack in x86_64?

A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp
Local Variables

• More generally, we can make space on the stack for N bytes by subtracting N from rsp
Local Variables

• More generally, we can make space on the stack for N bytes by subtracting N from rsp

• When we’re done, free the space by adding N back to rsp
Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What must a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries
Stack Frame Contents

• What needs to be stored in a stack frame?
  • Alternatively: What must a function know?

• Local variables
• Previous stack frame base address
• Function arguments
• Return value
• Return address

• Saved registers
• Spilled temporaries
Stack Frame Relationships

• If function 1 calls function 2:
  • function 1 is the **caller**
  • function 2 is the **callee**

• With respect to main:
  • main is the **caller**
  • function 1 is the **callee**

```
function 2 (callee)

function 1 (caller)

main

0xFFFFFFFF
```
Stack Frame Relationships

• If function 1 calls function 2:
  • function 1 is the **caller**
  • function 2 is the **callee**

• With respect to main:
  • main is the **caller**
  • function 1 is the **callee**
Where should we store all this stuff? Why?

- Previous stack frame base address
- Function arguments
- Return value
- Return address

A. In registers
B. On the heap
C. In the caller’s stack frame
D. In the callee’s stack frame
E. All of the above
F. None of the above
Calling Convention

• You could store this stuff wherever you want!
  • The hardware does NOT care
  • What matters: everyone agrees on where to find the necessary data

• Calling convention: agreed upon system for exchanging data between caller and callee

• When possible, keep values in registers
  • Accessing registers is faster than memory (stack)
x86_64 Calling Convention

• The function’s **return value**:  
  • In register %rax

• The caller’s %rbp value (caller’s **saved frame pointer**):  
  • Placed on the stack in the callee’s stack frame

• The **return address** (saved PC value to resume execution on return):  
  • Placed on the stack in the caller’s stack frame

• **Arguments** passed to a function:  
  • First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)  
  • Any additional arguments stored on the caller’s stack frame (shared with callee)
Top of the Stack

- Space for local & temporary vars, & saved register values
- saved %rbp (Caller’s stack frame)
- Return address (saved program counter)
- parameter 7 (first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
- ...
- parameter n
- ...
- ...

Bottom of Stack

- Earlier Stack Frames

Callee’s Frame or Active Frame (current frame in execution)

Caller’s Frame

Stack Pointer %rsp

Frame or Base Pointer %rbp

both caller & callee can access these:
push %rip (PC)
push input arguments to callee

lower memory address
call
higher memory address
return
x86_64 Calling Convention

• The function’s return value:
  • In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  • Placed on the stack in the callee’s stack frame

• The return address (saved PC value to resume execution on return)
  • Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  • First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  • Any additional arguments stored on the caller’s stack frame (shared with callee)
Return Value

• If the callee function produces a result, the caller can find it in %rax

• We saw this when we wrote our function in the lab:
  • Copy the result to %rax before we finishing up
x86_64 Calling Convention

• The function’s return value:
  • In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  • Placed on the stack in the callee’s stack frame

• The return address (saved PC value to resume execution on return)
  • Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  • First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  • Any additional arguments stored on the caller’s stack frame (shared with callee)
Dynamic Stack Accounting

- Dedicate CPU registers for stack bookkeeping
  - `%rsp` (stack pointer): Top of current stack frame
  - `%rbp` (frame pointer): Base of current stack frame

- Compiler maintains these pointers by inserting instructions on function call/return.
  - It doesn’t know (or care about) the exact addresses they point to.
  - This is why we’ve been accessing variables relative to `%rbp` in assembly...
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• Must adjust %rsp, rbp on call / return
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• Immediately upon calling a function:
  1. push %rbp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• Immediately upon calling a function:
  1. push %rbp
  2. Set %rbp = %rsp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• Immediately upon calling a function:
  1. pushl %rbp
  2. Set %rbp = %rsp
  3. Subtract N from %rsp

Callee can now execute.
Frame Pointer

- Must maintain invariant:
  - The current function’s stack frame is always between the addresses stored in `%rsp` and `%rbp`

- To return, reverse this:
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• To return, reverse this:
  1. set %rsp = %rbp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• To return, reverse this:
  1. set %rsp = %rbp
  2. pop %rbp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %rsp and %rbp

• To return, reverse this:
  1. set %rsp = %rbp
  2. pop %rbp

X86_64 has another convenience instruction for this: leaveq

Back to where we started.
Frame Pointer: Function Call

Initial state

push %rbp (store caller’s frame pointer)

mov %rsp, %rbp
(establish callee’s frame pointer)

sub $SIZE, %rsp
(allocate space for callee’s locals)
Frame Pointer: Function Return

Want to restore caller’s frame.

x86_64 provides a convenience instruction that does all of this: leaveq
x86_64 Calling Convention

• The function’s **return value**:  
  • In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)  
  • Placed on the stack in the callee’s stack frame

• The **return address** (saved PC value to resume execution on return)  
  • Placed on the stack in the caller’s stack frame

• Arguments passed to a function:  
  • First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)  
  • Any additional arguments stored on the caller’s stack frame (shared with callee)
funcA:
...
callq funcB
...

funcB:
push %rbp
mov %rsp, %rbp
...

Function B
Function A
...
Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

```
add $5, %rcx
```
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

```
add $5, %rcx
```

Update PC to next instruction.

Execute the `addl`.

---

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

mov $rcx, -8(%rbp)

Text Memory Region

funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

\[
\text{mov } \$\text{rcx, } -8(\%\text{rbp})
\]

Update PC to next instruction.

Execute the mov.
Program Counter

Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Keep executing in a straight line downwards like this until:

We hit a jump instruction.
We call a function.

Text Memory Region

funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
Changing the PC: Jump

• On a (non-function call) jump:
  – Check condition codes
  – Set PC to execute elsewhere (not next instruction)

• Do we ever need to go back to the instruction after the jump?

  Maybe (and if so, we’d have a label to jump back to), but usually not.
Changing the PC: Functions

What we’d like this to do:

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Changing the PC: Functions

What we’d like this to do:

Set up function B’s stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Changing the PC: Functions

What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %rax).

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Changing the PC: Functions

What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %rax).

Restore function A’s stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
What we’d like this to do:

Return:
Go back to what we were doing before funcB started.

Unlike jumping, we intend to go back!
Like `push`, `pop`, and `leave`, `call` and `ret` are convenience instructions. What should they do to support the PC-changing behavior we need? (The PC is `%rip`.)

```
call         ret
In words:    In words:
In instructions:  In instructions:
```
Functions and the Stack

Executing instruction:
callq funcB

PC points to next instruction

Text Memory Region

funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...  
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...  
mov $10, %rax
leaveq
retq
Functions and the Stack

1. push %rip

Text Memory Region

```assembly
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)

Text Memory Region

funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
... 
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
Functions and the Stack

6. (resume funcA)

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
6. (resume funcA)

Text Memory Region

```
funcA:
  add $5, %rcx
  mov %rcx, -8(%rbp)
...
  callq funcB
  add %rax, %rcx
...

funcB:
  push %rbp
  mov %rsp, %rbp
...
  mov $10, %rax
  leaveq
  retq
```
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
6. (resume funcA)

Return address:
Address of the instruction we should jump back to when we finish (return from) the currently executing function.
## x86_64 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
</table>
| **push**    | Create space on the stack and place the source there. | push $8, %rsp  
 mov src, (%rsp) |
| **pop**     | Remove the top item off the stack and store it at the destination. | mov (%rsp), dst  
 add $8, %rsp |
| **callq**   | 1. Push return address on stack  
 2. Jump to start of function | push %rip  
 jmp target |
| **leaveq**  | Prepare the stack for return (restoring caller’s stack frame) | mov %rbp, %rsp  
 pop %rbp |
| **retq**    | Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip)) | pop %rip |
x86_64 Calling Convention

• The function’s return value:
  • In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  • Placed on the stack in the callee’s stack frame

• The return address (saved PC value to resume execution on return)
  • Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  • First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  • Any additional arguments stored on the caller’s stack frame (shared with callee)
Function Arguments

• Most functions don’t receive more than 6 arguments, so x86_64 can simply use registers most of the time.

• If we do have more than 6 arguments though (e.g., perhaps a printf with lots of placeholders), we can’t fit them all in registers.

• In that case, we need to store the extra arguments on the stack. By convention, they go in the caller’s stack frame.
If we need to place arguments in the caller’s stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn’t matter

D. Somewhere else
# x86_64 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Create space on the stack and place the source there.</td>
<td><code>sub $8, %rsp</code>&lt;br&gt;<code>mov src, (%rsp)</code></td>
</tr>
<tr>
<td>pop</td>
<td>Remove the top item off the stack and store it at the destination.</td>
<td><code>mov (%rsp), dst</code>&lt;br&gt;<code>add $8, %rsp</code></td>
</tr>
<tr>
<td>callq</td>
<td>1. Push return address on stack 2. Jump to start of function</td>
<td><code>push %rip</code>&lt;br&gt;<code>jmp target</code></td>
</tr>
<tr>
<td>leaveq</td>
<td>Prepare the stack for return (restoring caller’s stack frame)</td>
<td><code>mov %rbp, %rsp</code>&lt;br&gt;<code>pop %rbp</code></td>
</tr>
<tr>
<td>retq</td>
<td>Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))</td>
<td><code>pop %rip</code></td>
</tr>
</tbody>
</table>
Arguments

• Extra arguments to the callee are stored just underneath the return address.

• Does it matter what order we store the arguments in?

• Not really, as long as we’re consistent (follow conventions).

This is why arguments can be found at positive offsets relative to %rbp.
Top of the Stack

- Space for local & temporary vars, & saved register values
- saved %rbp (Caller’s stack frame)
- Return address (saved program counter)
  - parameter 7 (first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
  - ...
  - parameter n

Callee’s Frame or Active Frame (current frame in execution)
- Call

Caller’s Frame
- Push input arguments to callee
- Call

Bottom of Stack

Earlier Stack Frames
- Return

Stack Pointer %rsp
- Lower memory address

Frame or Base Pointer %rbp
- Higher memory address
Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries

```
function 2
function 1
main
```
0xFFFFFFFF
Saving Registers

• Registers are a relatively scarce resource, but they’re fast to access. Memory is plentiful, but slower to access.

• Should the caller save its registers to free them up for the callee to use?
• Should the callee save the registers in case the caller was using them?
• Who needs more registers for temporary calculations, the caller or callee?

• Clearly the answers depend on what the functions do...
Splitting the difference...

• We can’t know the answers to those questions in advance...

• Divide registers into two groups:
  • Caller-saved: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
    • If the caller wants to preserve these registers, it must save them prior to calling callee
    • callee free to trash these, caller will restore if needed
  • Callee-saved: %rbx, %r12, %r13, %r14, %r15
    • If the callee wants to use these registers, it must save them first, and restore them before returning
    • caller can assume these will be preserved
Running Out of Registers

• Some computations require more than 16 general-purpose registers to store temporary values.

• *Register spilling*: The compiler will move some temporary values to memory, if necessary.
  • Values pushed onto stack, popped off later
  • No explicit variable declared by user
  • This is getting to the boundary of CS 31 information – take CS 75 (compilers) for more details.
Up next...

- Connecting Arrays, Structs, and Pointers with assembly