Recursion & Critical Reading

CSE 351 Summer 2021

Instructor:

Mara Kirdani-Ryan

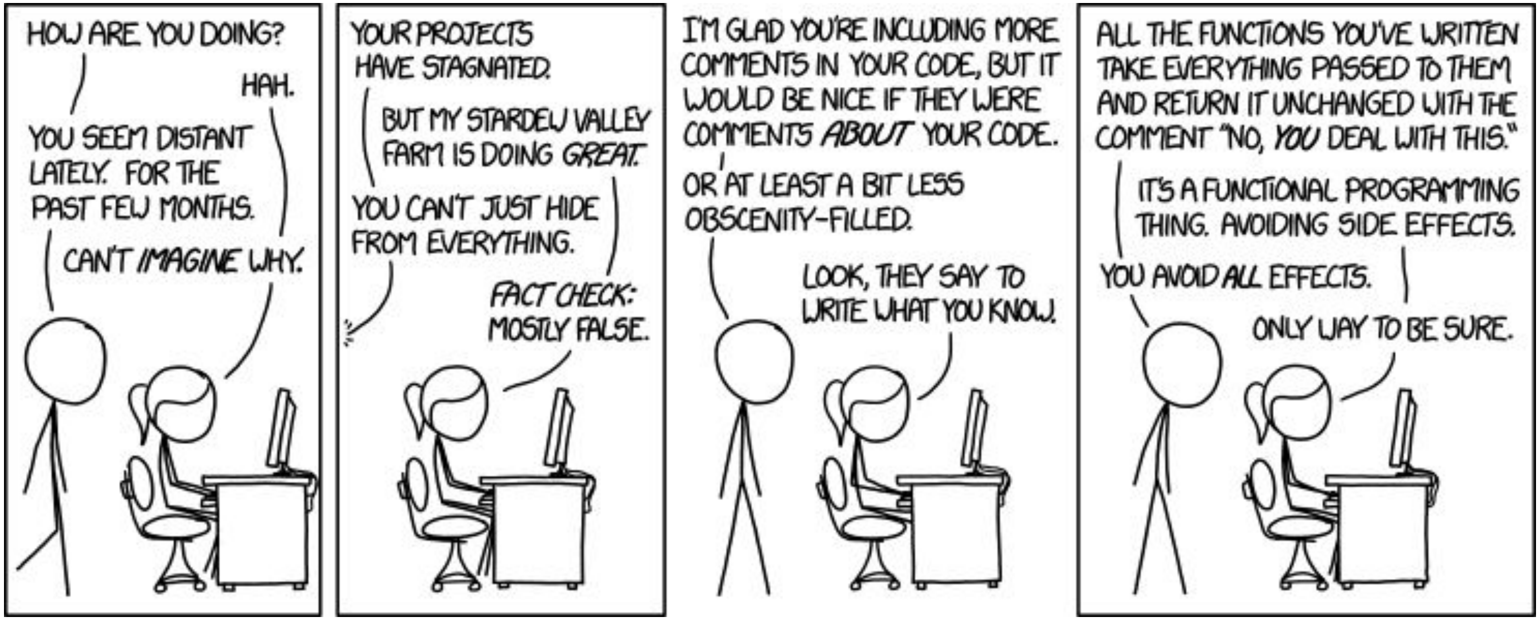
Teaching Assistants:

Kashish Aggarwal

Nick Durand

Colton Jobes

Tim Mandzyuk



http://xkcd.com/1790/

Gentle, Loving Reminders

- Mid-quarter Survey **due tonight** (7/16) -- 8pm
  - Submit via Canvas!

- hw10 due tonight, hw11 due Monday
- Lab 2 due Wednesday (7/21)
  - GDB Tutorial on Gradescope walks through first phase

- Creativity takes time & space! Think about US#2!
  - But, only if there's space!
  - I'm going to try to have feedback on US#1 by Monday
    - Thanks for your effort!

Disclaimer:
I’m having a hard time!

I’m doing what I can, you’re responsible for your own learning.

Learning Objectives

Understanding this lecture means you can:

- Trace register usage through a function call
- Trace callee/caller register usage through a recursive function call/return
- Perform a critical reading of the introduction to our textbook, analyzing for assumptions and values
- Perform a critical reading of the reasons that you took this course, analyzing for assumptions and values

Example: `increment`

```
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

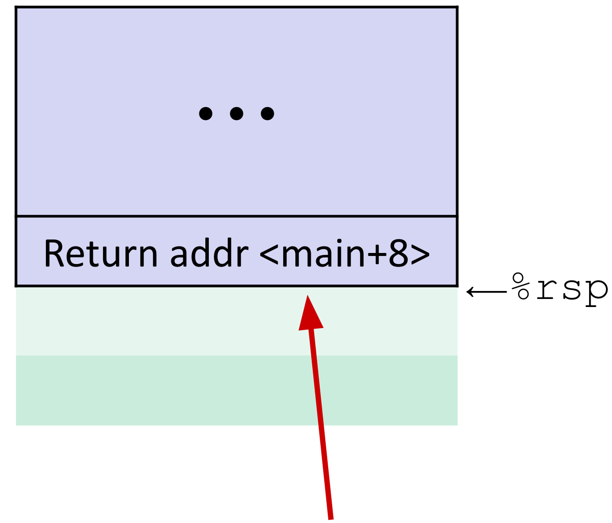
Register	Use(s)
%rdi	1st arg (p)
%rsi	2nd arg (val), y
%rax	x, return value

Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack



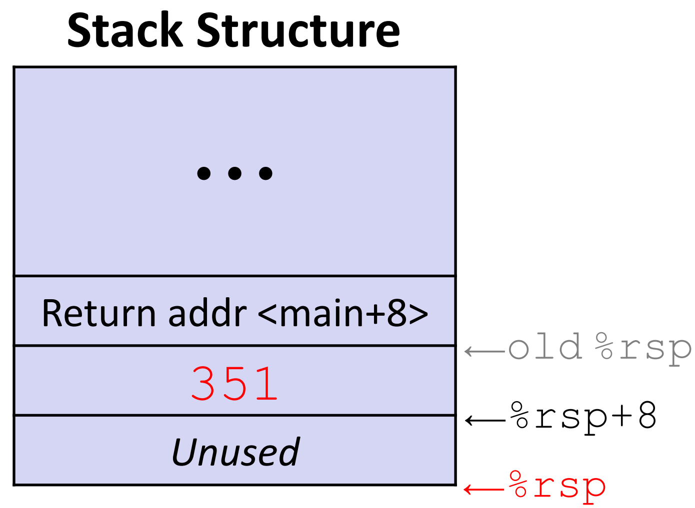
- Return address on stack is the address of instruction immediately *following* the call to "`call_incr`"
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Allocate space for local vars



- Setup space for local variables
  - Only `v1` needs stack space
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

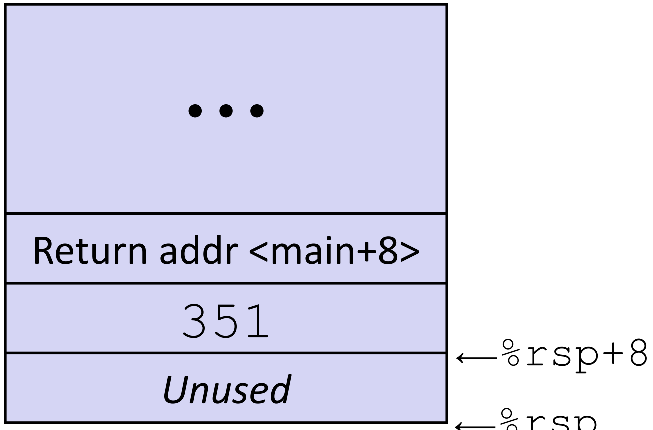
Procedure Call Example (step 2)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $351, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Set up parameters for call to `increment`

Stack Structure



Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`100`

Aside: `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Stack Structure

• • •
Return addr <main+8>
351
Unused
Return addr <call_incr+?> ← %rsp

- State while inside `increment`
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

Register	Use(s)
%rdi	&v1
%rsi	100
%rax	

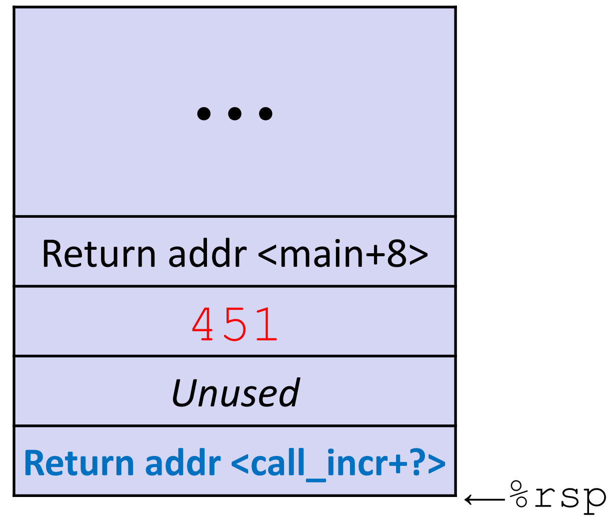
Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax # x = *p
  addq    %rax, %rsi   # y = x + 100
  movq    %rsi, (%rdi) # *p = y
  ret
```

Stack Structure



- State while inside `increment`
  - *After* code in body has been executed

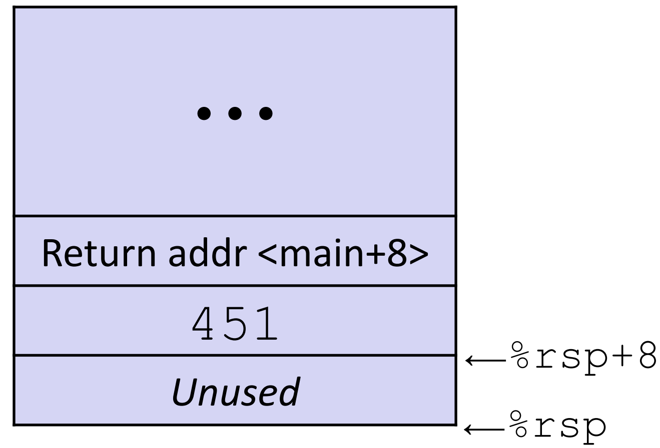
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

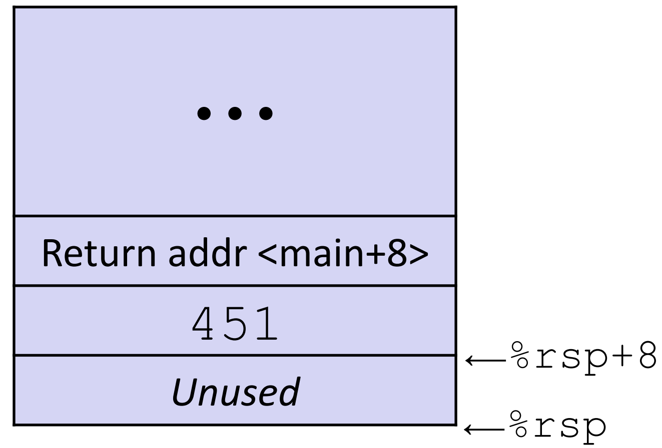
Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← Update `%rax` to contain `v1+v2`

Stack Structure



Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`451`
`%rax`	`451+351`

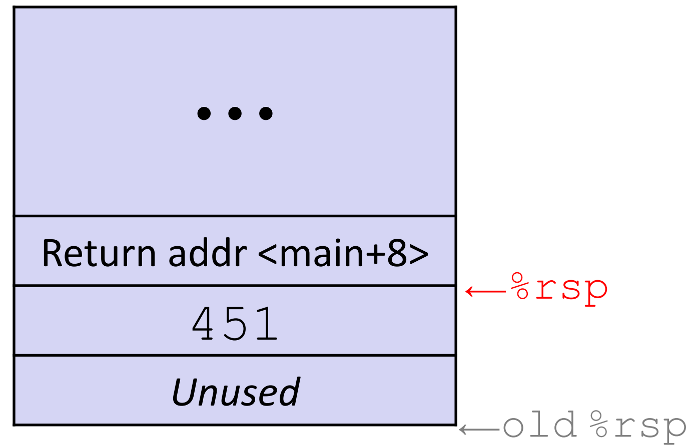
Procedure Call Example (step 7)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← De-allocate space for local vars (addq $16, %rsp)

Stack Structure



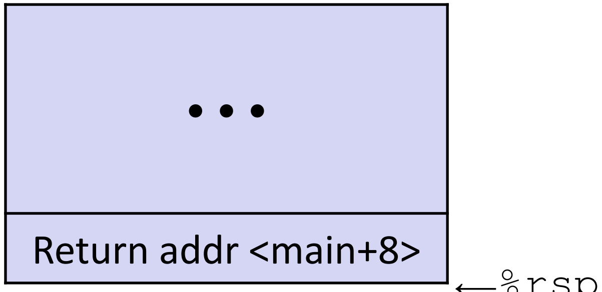
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



- State *just before* returning from call to `call_incr`

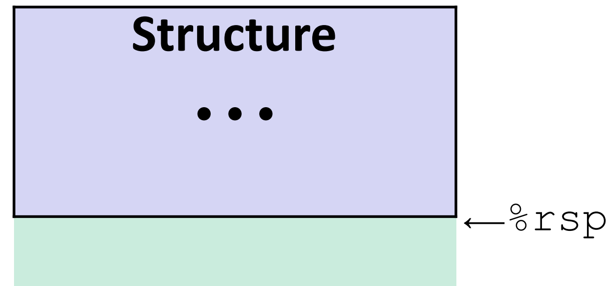
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Final Stack Structure



- State immediately *after* returning from call to `call_incr`
  - Return addr has popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`451`
`%rax`	`802`

Feelings check: Procedure calls?

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- **Register Saving Conventions**
- Illustration of Recursion

Register Saving Conventions

- When procedure `whoa` calls `who`:
  - `whoa` is the ***caller***
  - `who` is the ***callee***
- Can registers be used for temporary storage?

```
whoa:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

- No! Contents of register `%rdx` overwritten by `who`!
- This could be trouble – something should be done. Either:
  - ***Caller*** should save `%rdx` before the call (and restore it after the call)
  - ***Callee*** should save `%rdx` before using it (and restore it before returning)

Register Saving Conventions

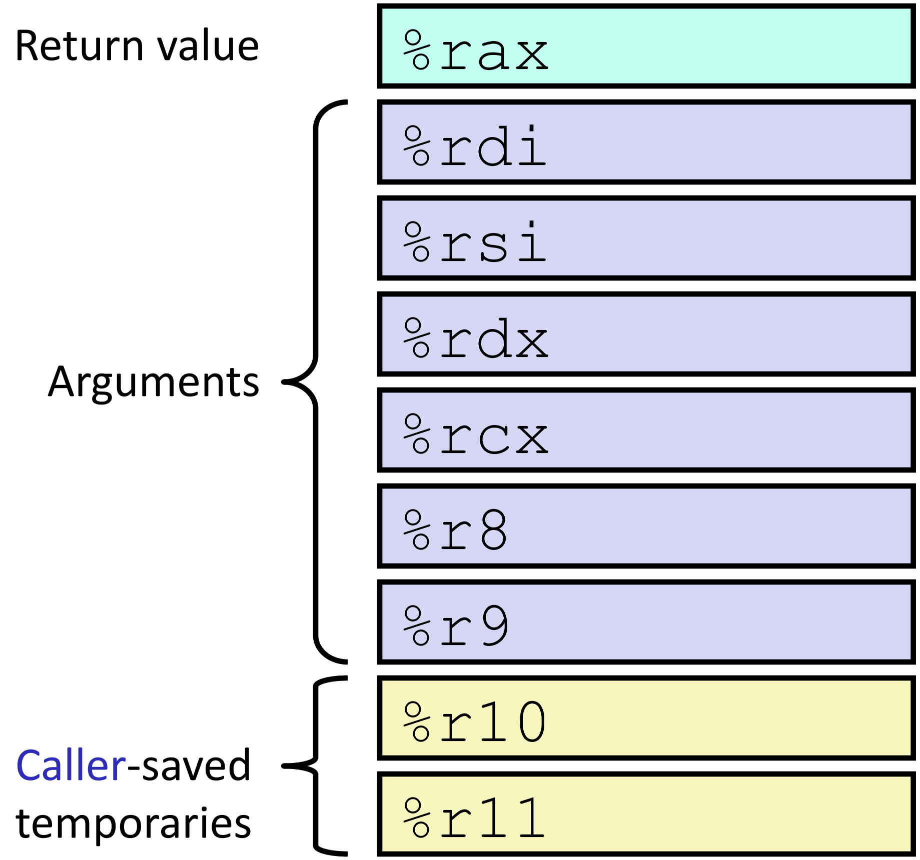
- ***"Caller-saved" registers***
  - It is the **caller**'s responsibility to save any important data in these registers before calling another procedure (*i.e.* the **callee** can freely change data in these registers)
  - **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call

Register Saving Conventions

- ***"Callee-saved" registers***
  - It is the callee's responsibility to save any data in these registers before using the registers (*i.e.* the **caller** assumes the data will be the same across the **callee** procedure call)
  - **Callee** saves values in its stack frame before using, then restores them before returning to **caller**

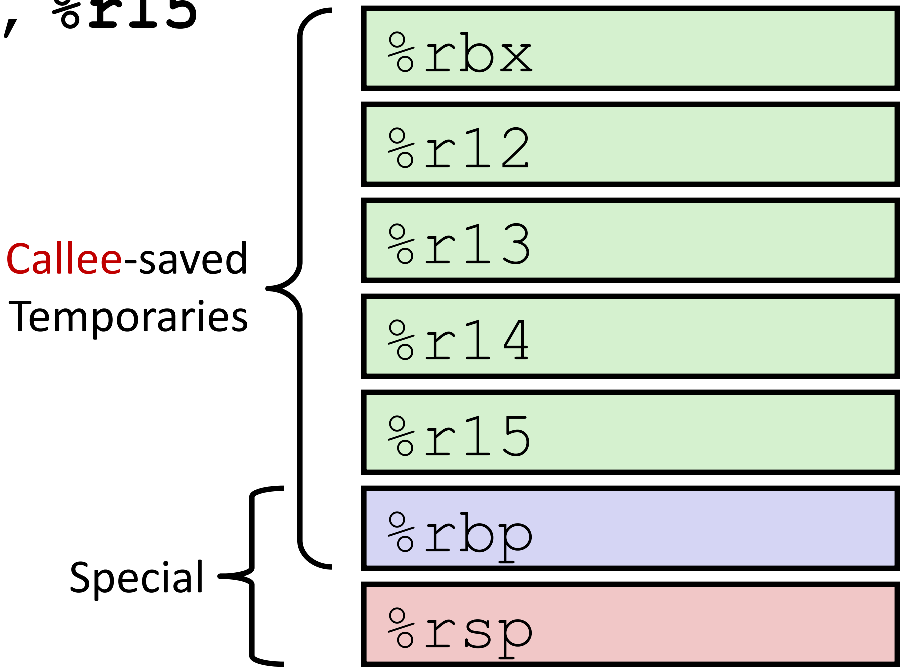
x86-64 Linux Register Usage, part 1

- **`%rax`**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%rdi, ..., %r9`**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%r10, %r11`**
  - **Caller**-saved & restored
  - Can be modified by procedure



x86-64 Linux Register Usage, part 2

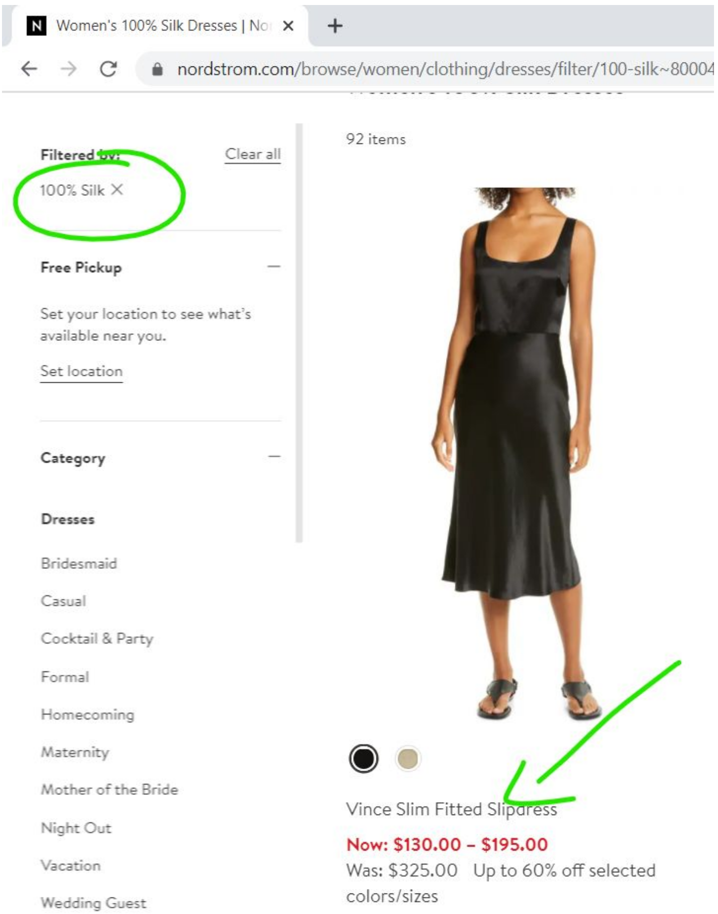
- **`%rbx, %r12, %r13, %r14, %r15`**
  - **Callee**-saved
  - **Callee** must save & restore
- **`%rbp`**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer
  - Can mix & match
- **`%rsp`**
  - Special form of **callee** save
  - Restored to original value upon exit from procedure

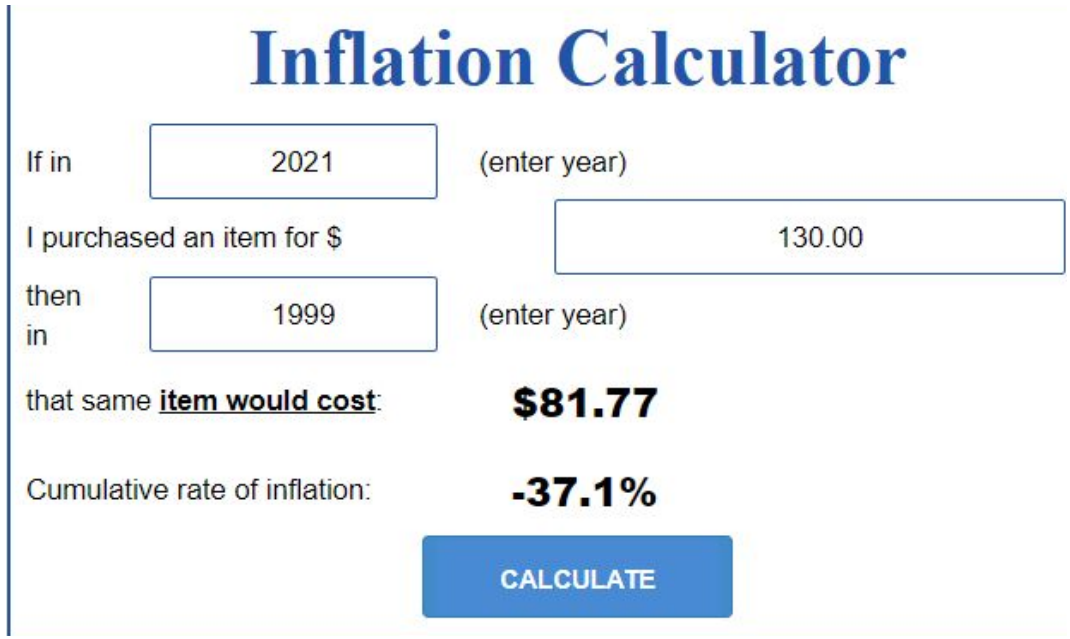


x86-64 64-bit Registers: Usage Conventions

`%rax`	Return value - Caller saved	`%r8`	Argument #5 - Caller saved
`%rbx`	Callee saved	`%r9`	Argument #6 - Caller saved
`%rcx`	Argument #4 - Caller saved	`%r10`	Caller saved
`%rdx`	Argument #3 - Caller saved	`%r11`	Caller Saved
`%rsi`	Argument #2 - Caller saved	`%r12`	Callee saved
`%rdi`	Argument #1 - Caller saved	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

Wait, $89??? (credit to Kimi Locke)

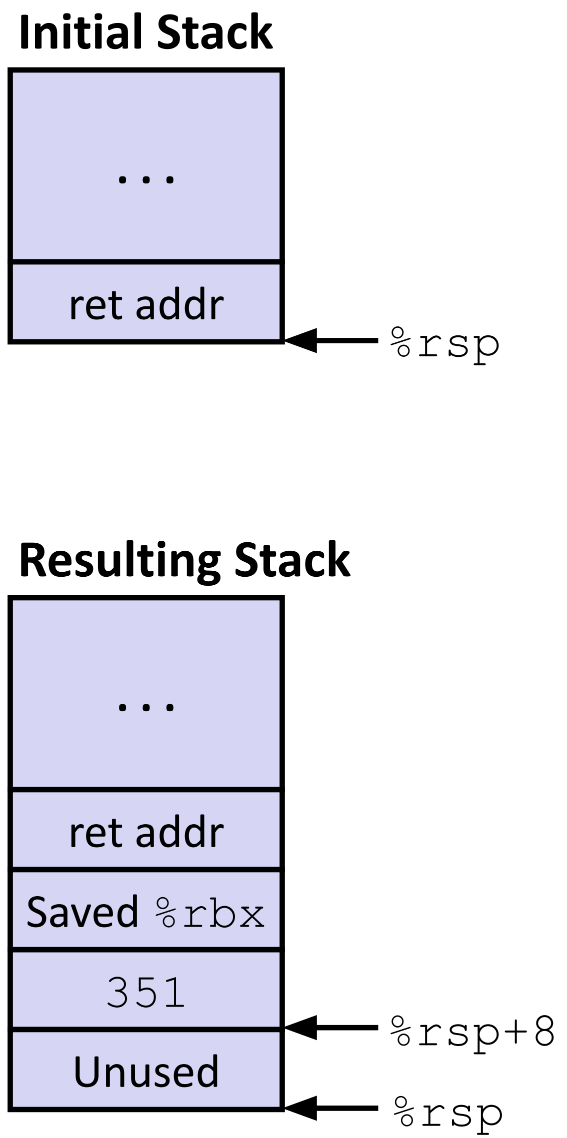




Callee-Saved Example (step 1)

```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

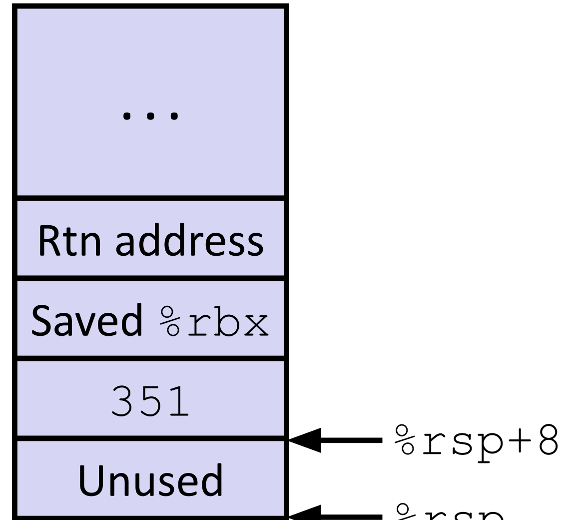


Callee-Saved Example (step 2)

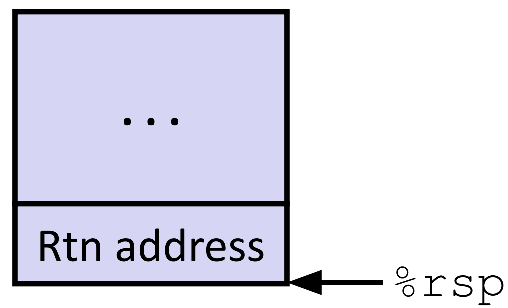
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Pre-return Stack



Why Caller *and* Callee Saved?

- “**Efficiency**”
- We want *one* calling convention to simply separate implementation details between caller and callee

- In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better
  - If callee doesn’t need a register, callee-save is better
  - If “do need to save”, callee-save generally makes smaller programs
    - Functions are called from multiple places

- So… “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring

Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values
- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- **Illustration of Recursion**

Recursive Function

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

Compiler Explorer:
https://godbolt.org/z/xFCrsw

- Compiled with `-O1` for brevity instead of `-Og`
- Try `-O2` instead!

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

Recursive Function: Base Case

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

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

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

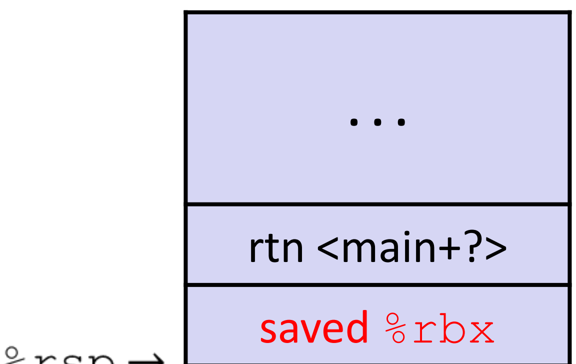
Trick because some AMD hardware doesn't like jumping to `ret`

Recursive Function: Callee Reg Save

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

Register	Use(s)	Type
`%rdi`	`x`	Argument

The Stack



Need original value of `x` *after* recursive call to `pcount_r`.

"Save" by putting in `%rbx` (**callee** saved), but need to save old value of `%rbx` before you change it.

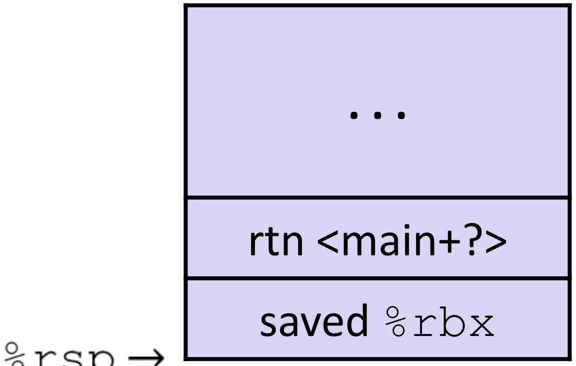
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

Recursive Function: Call Setup

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

Register	Use(s)	Type
%rdi	x (new)	Argument
%rbx	x (old)	Callee saved

The Stack




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

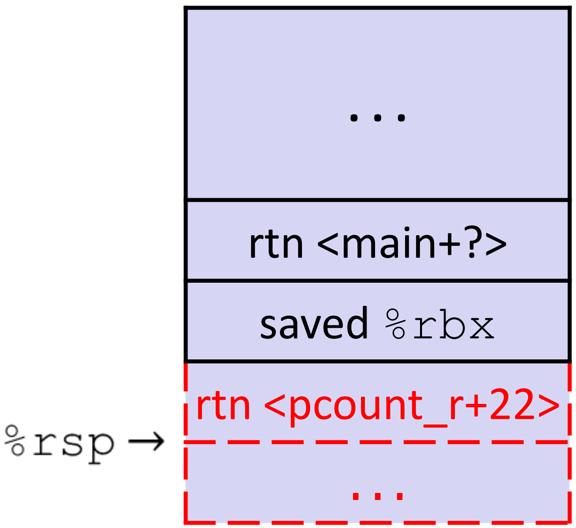
Recursive Function: Call

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

Register	Use(s)	Type
%rax	Recursive call return value	Return value
%rbx	x (old)	Callee saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

The Stack



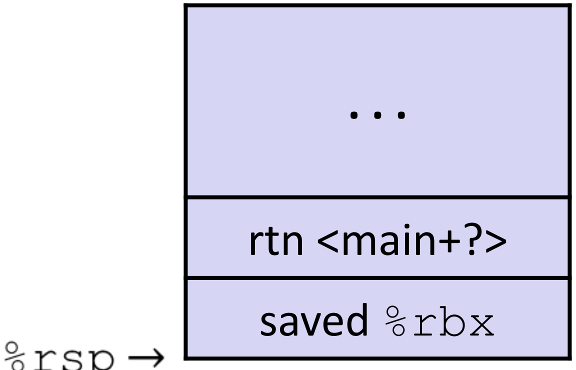
Recursive Function: Result

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

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

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

The Stack



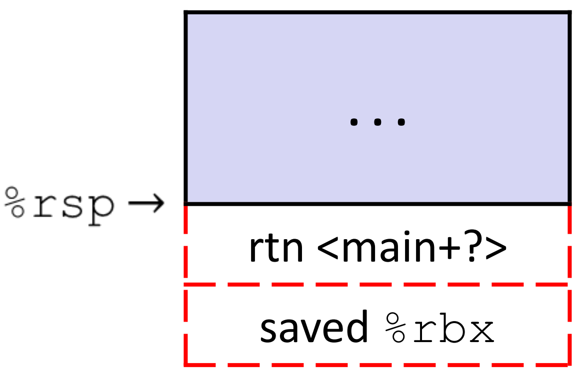
Recursive Function: Completion

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

Register	Use(s)	Type
`%rax`	Return value	Return value
`%rbx`	Previous `%rbx` value	Callee restored

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

The Stack



Observations About Recursion

- Works without any special consideration
  - Stack frames: each function call has private storage
    - Saved registers & local variables, return address
    - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does so (*e.g.* buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion
  - (P calls Q; Q calls P)

x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

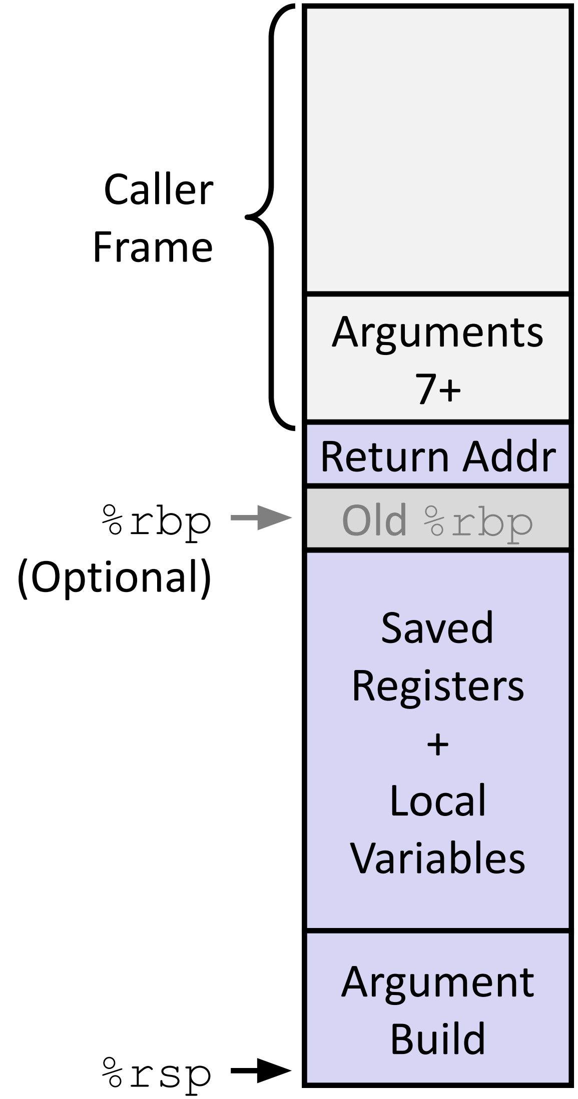
x86-64 Stack Frames

- Procedures *needs* to grow stack frames when:
  - Has too many local variables to hold in **caller**-saved registers
  - Has local variables that are arrays or structs
  - Uses `&` to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using **caller**-saved registers and then calls a procedure
  - Modifies/uses **callee**-saved registers

Feelings Check: Recursion!

x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of *instructions and conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack



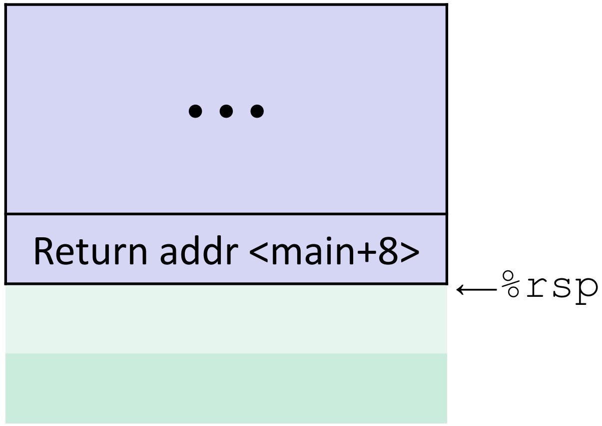
Procedure Call Example – Handout

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Stack Structure



Register	Use/Value(s)
%rdi	
%rsi	
%rax	

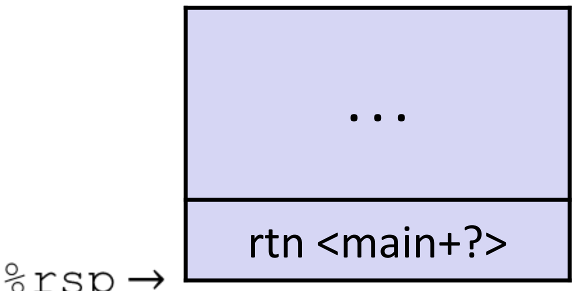
Recursive Function – Handout

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

Register	Use(s)	Type
%rax	Recursive call return value	Return value
%rbx	x (old)	Callee saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

The Stack



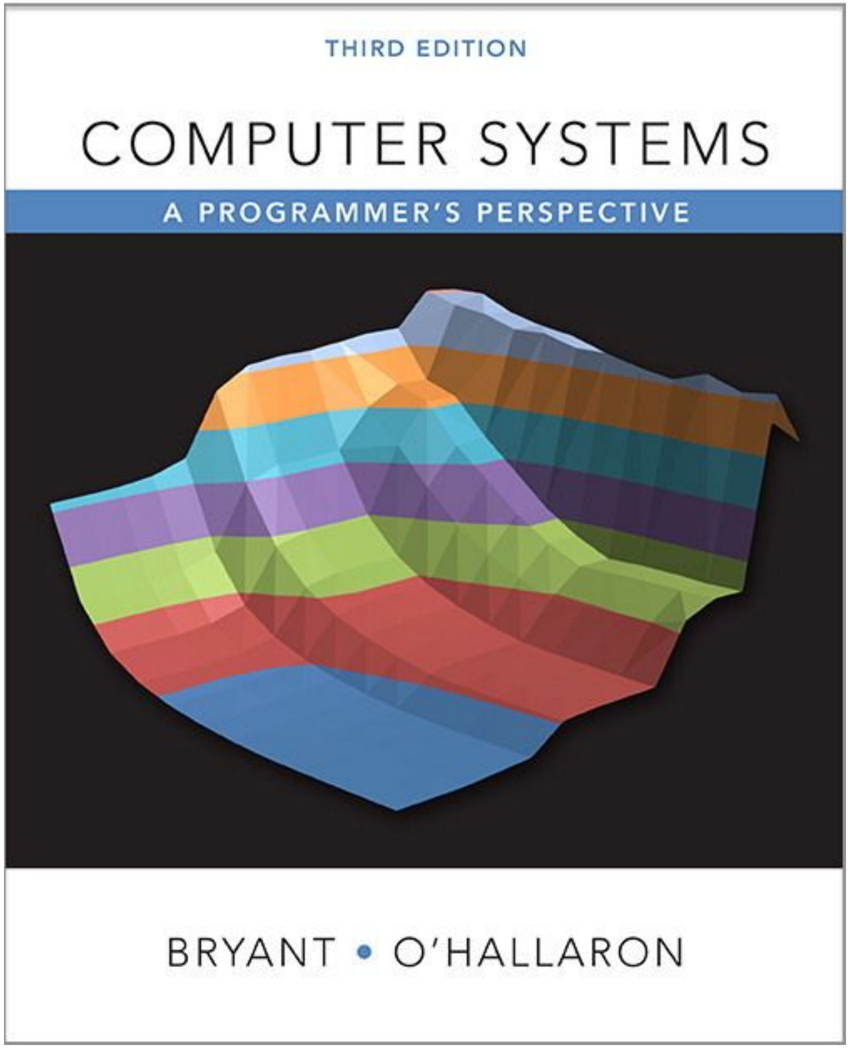
Textbook: critical reading

Textbook: critical reading

An experiment, if that wasn't clear

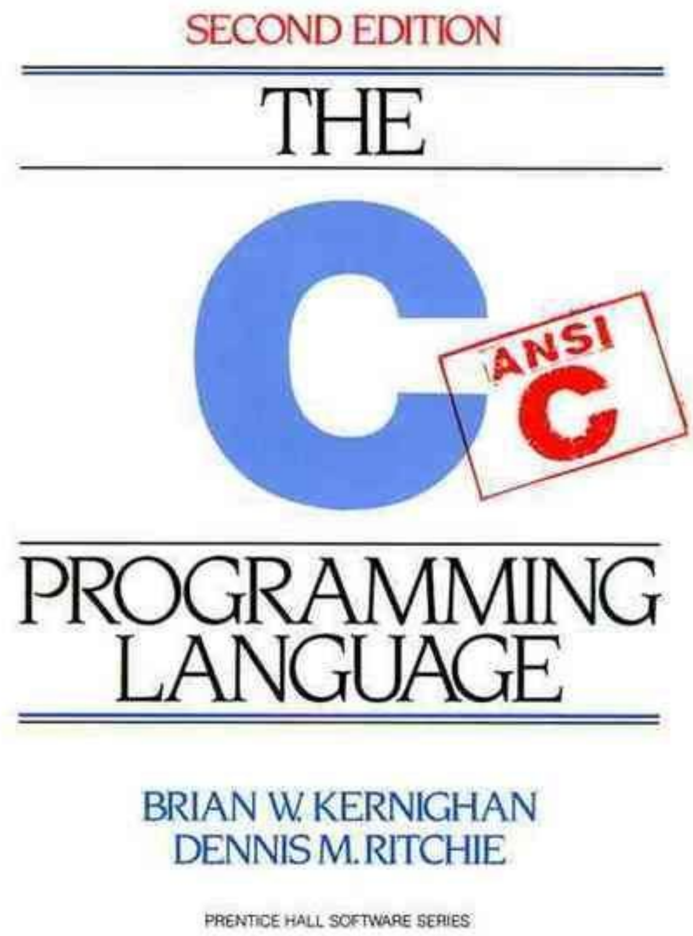
Hopefully y'all did the reading!

- Developed @ CMU
  - "Intro to Computer Systems"
- Textbook followed from course
- Generally used for computer systems courses in CS
  - EE/ECE might have more HW detail



Hopefully y'all did the reading!

- Kernighan & Ritchie
  - "standardized" C language
- Short, ~200 pages
- "Seminal" C book
- 1978, 1988 editions



Breakouts!
What’d you see?

Thoughts on course objectives? Any ideologies?

What I noted, among other things

Our aim in 15-213/18-213/15-513 is to help you become a better programmer by teaching you the basic concepts underlying all computer systems. We want you to learn what really happens when your programs run, so that when things go wrong (as they always do) you will have the intellectual tools to solve the problem.

Why do you need to understand computer systems if you do all of your programming in high level languages? In most of computer science, we're pushed to make abstractions and stay within their frameworks. But, any abstraction ignores effects that can become critical. As an analogy, Newtonian mechanics ignores relativistic effects. The Newtonian abstraction is completely appropriate for bodies moving at less than $0.1c$, but higher speeds require working at a greater level of detail.

2. *You've got to know assembly language.* Even if you never write programs in assembly, The behavior of a program cannot be understood sometimes purely based on the abstraction of a high-level language. Further, understanding the effects of bugs requires familiarity with the machine-level model.

4. *There is more to performance than asymptotic complexity.* Constant factors also matter. There are systematic ways to evaluate and improve program performance.

Thoughts the CS:APP textbook?
Any ideologies?

What I noted, among other things

This book (known as CS:APP) is for computer scientists, computer engineers, and others who want to be able to write better programs by learning what is going on "under the hood" of a computer system.

Our aim is to explain the enduring concepts underlying all computer systems, and to show you the concrete ways that these ideas affect the correctness, perfor-

of these aspects, with the unifying theme of a programmer's perspective.

If you study and learn the concepts in this book, you will be on your way to becoming the rare *power programmer* who knows how things work and how to fix them when they break. You will be able to write programs that make better use of the capabilities provided by the operating system and systems software.

grammer, the compiler, and the operating system can take to reduce these threats. Learning the concepts in this chapter helps you become a better programmer, because you will understand how programs are represented on a machine. One certain benefit is that you will develop a thorough and concrete understanding of pointers.

language, and it is clearly and beautifully described in the classic "K&R" text by Brian Kernighan and Dennis Ritchie [61]. Regardless of your programming background, consider K&R an essential part of your personal systems library. If your prior experience is with an interpreted language, such as Python, Ruby, or

ations.

Having a solid understanding of computer arithmetic is critical to writing reliable programs. For example, programmers and compilers cannot replace the expression (x<y) with (x-y < 0), due to the possibility of overflow.

Extended notables

unsigned numbers. We cover the mathematical properties of arithmetic operations. Novice programmers are often surprised to learn that the (two's-complement) sum or product of two positive numbers can be negative. On the other hand, two's-complement arithmetic satisfies many of the algebraic properties of integer arithmetic, and hence a compiler can safely transform multiplication by a constant into a sequence of shifts and adds. We use the bit-level operations of C to demonstrate the principles and applications of

the other hand, most students, including all computer scientists and computer engineers, would be required to use and program computers on a daily basis. So we decided to teach about systems from the point of view of the programmer, using the following filter: we would cover a topic only if it affected the performance, correctness, or utility of user-level C programs.

For example, topics such as hardware adder and bus designs were out. Top-

Chapter 5: Optimizing Program Performance. This chapter introduces a number of techniques for improving code performance, with the idea being that programmers learn to write their C code in such a way that a compiler can then

of programs containing memory referencing errors such as storage leaks and invalid pointer references. Finally, many application programmers write their own storage allocators optimized toward the needs and characteristics of the application. This chapter, more than any other, demonstrates the benefit of covering both the hardware and the software aspects of computer systems in a unified way. Traditional computer architecture and operating systems texts present only part of the virtual memory story.

Thoughts on the K&R textbook?

What I noted, among other things:

and a rich set of operators. C is not a ‘‘very high level’’ language, nor a ‘‘big’’ one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up the language, although access to more knowledgeable colleague will help.

In our experience, C has proven to be a pleasant, expressive and versatile language for a wide variety of programs. It is easy to learn, and it wears well as on’s experience with it grows. We hope that this book will help you to use it well.

C is a relatively ‘‘low-level’’ language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

Although the absence of some of these features may seem like a grave deficiency, (‘‘You mean I have to call a function to compare two character strings?’’), keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax

Subtle, but not invisible

Themes

- Performance is really, really important
- Simplicity is better, especially with regards to performance
- "Performance, Correctness, Utility"
- *"Rare Power Programmers"* understand the entire system
  - In reality, no one understands the entire system
- Only "novices" are surprised by overflow, or compare floats for equality
- C is "essential" for systems programmers (this is kind of true, but self-fulfilling)

We’ve seen this all before!

Though, maybe not so close to home!

We've seen this all before!

Though, maybe not so close to home!

Let's go even closer!

Why are you here?

Why'd you take this class?

- I took this class as an undergrad because it was required…
- Though, I had so much fun that I ended up staying in computer systems/security
  - Lab 2 was my favorite ✨ ✨
- Looking through the survey, I found some similarities

Breakouts!

Why'd you take this course?
What did you uncover with a critical reading?

Asking Questions -- from y'all

- **"I want to be a better programmer"**
  - What does "better" mean?
  - Why is it important to you to be a better programmer?
- **"I want to learn how to program in C"**
  - Why is it important to learn C programming?
- **"I want to understand core computing concepts"**
  - Why is 351 "core"? Because we said so? Because the Allen School said so?

These are entirely reasonable, also.

I'd just like y'all to understand yourselves a bit better!

My goal, when I teach, is to off the opportunity for you to learn something that's broadly applicable, regardless of where you end up.

Self-discovery, by another name.

Future Employers...

- I mean, y'all need jobs, I get it
- Most CS employers will replicate historic computing values
  - Efficiency
  - Performance
  - Minimalism (or "elegance", you might hear it this way)
- Your career isn't defined by your first job!
  - Most of you will do more than one thing!
  - Asking "Why" helps you learn about yourself!
  - Why Big Four? Why Microsoft?

These are reflective questions.

You might need time and space to answer them.

When you answer "why", who's answering?

Answering Why

- *"I'm taking 351 because I want to be a better programmer"*
- **"Understanding the underlying system makes you better at debugging and understanding performance"**
  - Why is it important to be good at debugging?
  - Why is it important to understand performance?
  - Why is it important to understand the system?

CS has an ideology!

The Allen School is no exception.

Most ideology is unexamined!

It’s like ___, most folks will probably only look if something’s going wrong.

This is true of ideology broadly!

If we don’t ask questions, we’re doomed to replicate what we’ve been taught.

What I was taught in CS

- I should understand everything, all the way down
- I should challenge myself in courses, at the expense of my self, and my relationships
- *Rare Power Programmers (i.e. 10x programmers)* are real, and I should try to be one
  - By working myself as hard as possible, obviously
- If I get a job at Google/MS/FB/Apple/Amazon, I'm successful, I should be embarrassed otherwise
  - Some might be relevant at UW, y'all know better than me

Try to always question what you're learning!

This helped me figure myself out.

Asking for help

- Come to us with “why” questions!
  - We’re happy to ask more questions
  - We’re happy to give historical context
  - We’re happy to sift through pieces to get to ideology
- This extends well beyond this course!
  - Don’t stop asking, especially if it’s “off-topic”