CSCI 331:
Introduction to Computer Security

Lecture 12: How C functions work

Instructor: Dan Barowy

Williams

Topics

Practice midterm review?

How C functions work

Your to-dos

1. Project part 2, **due Sunday 10/22**.
2. Read and take notes (Miller) **for Thur 10/26**.
3. Lab 5, **due Sunday 10/29**.

Dunning-Kruger Effect

A **cognitive bias** in which people mistakenly assess their cognitive ability as **greater than it is**.

Journal of Personality and Social Psychology
1999, Vol. 77, No. 6, 1121–1134

0022-3514/99/$3.00

Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments

Justin Kruger and David Dunning
Cornell University

People tend to hold overly favorable views of their abilities in many social and intellectual domains. The authors suggest that this overestimation occurs, in part, because people who are unskilled in these domains suffer a dual burden: Not only do these people reach erroneous conclusions and make unfortunate choices, but their incompetence robs them of the metacognitive ability to realize it. Across 4 studies, the authors found that participants scoring in the bottom quartile on tests of humor, grammar, and logic grossly overestimated their test performance and ability. Although their test scores put them in the 12th percentile, they estimated themselves to be in the 62nd. Several analyses linked this miscalibration to deficits in metacognitive skill, or the capacity to distinguish accuracy from error. Paradoxically, improving the skills of participants, and thus increasing their metacognitive competence, helped them recognize the limitations of their abilities.

to deficits in metacognitive skill, or the capacity to distinguish accuracy from improving the skills of participants, and thus increasing their metacognitive com recognize the limitations of their abilities.

> It is one of the essential features of such incompetence that the person so afflicted is incapable of knowing that he is incompetent. To have such knowledge would already be to remedy a good portion of the offense. (Miller, 1993, p. 4)

In 1995, McArthur Wheeler walked into two Pittsburgh banks and robbed them in broad daylight, with no visible attempt at disguise. He was arrested later that night, less than an hour after videotapes of him taken from surveillance cameras were broadcast on the 11 o'clock news. When police later showed him the surveillance tapes, Mr. Wheeler stared in incredulity. "But I wore the juice," he mumbled. Apparently, Mr. Wheeler was under the impression that rubbing one's face with lemon juice rendered it invisible to videotape cameras (Fuocco, 1996).

We bring up the unfortunate affairs of Mr. Wheeler to make three points. The first two are noncontroversial. First, in many domains in life, success and satisfaction depend on knowledge, wisdom, or savvy in knowing which rules to follow and which strategies to pursue. This is true not only for committing crimes, but also for many tasks in the social and intellectual domains, such

as promoting effective le
solid logical argument,
study. Second, people di
gies they apply in these
berg, 1989; Dunning, P
1998), with varying level
theories that people apply
favorable results. Other
McArthur Wheeler, are
competent, or dysfunctio

Perhaps more controve
focus of this article. We a
the strategies they adopt
suffer a dual burden: Not
and make unfortunate cho
the ability to realize it. In
the mistaken impression
(1993) perceptively obse
and as Charles Darwin
"ignorance more frequer
edge" (p. 3).

In essence, we argue th
a particular domain are
evaluate competence in t

Justin Kruger and David Dunning, Department of Psychology, Cornell University.

We thank Betsy Ostrov, Mark Stalnaker, and Boris Veysman for their

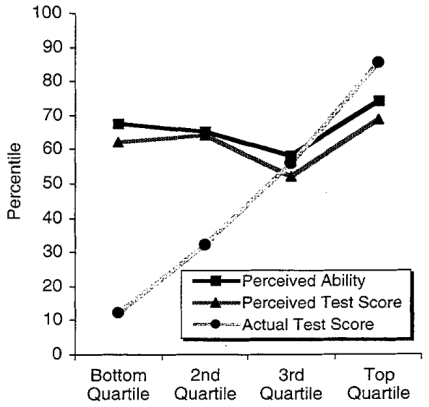


Figure 2. Perceived logical reasoning ability and test performance as a function of actual test performance (Study 2).

"20-item logical reasoning test that we created using questions taken from a Law School Admissions Test (LSAT) test preparation guide"

Dunning-Kruger Effect: Security Implications

Thinking that you have more ability than you do is a **security vulnerability**.

An **incompetent security audit** may **leave important parts of your system undefended**.

Countermeasures? **Do what Stoll does:**

- Have a "beginner's mind." What **do** you know for sure? What **don't** you know? Be honest.
- **Seek external validation** of both **facts** and your **abilities**.
- It's fine if you don't know something **as long as you know you don't know**. But **then learn it thoroughly**.

Midterm Study Guide Solutions?

ARM

Why am I learning this?

Wise words from my favorite philosopher:

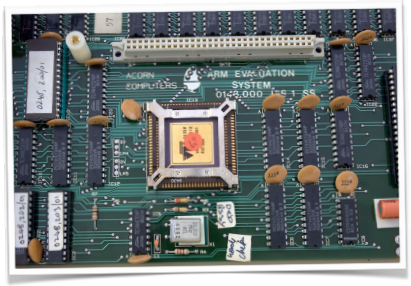
"I find it hard to remember things
I don't give a crap about." —House, M.D.

Why you should "give a crap" about assembly:

It's key to understanding control flow integrity attacks and defenses.

ARM

The ARM *instruction set architecture* is a family of microprocessors initially introduced in 1985.



We will focus on a 32-bit version, ARMv6, in this class. ARMv8 added 64 bit instructions, and the CPU in your cellphone is very likely to be a related architecture.

Instruction Set Architecture

An **instruction set architecture** (**ISA**) is an **abstraction** of a computer processor, much in the same way that an **interface** is an abstraction of a Java **class**.

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set

You can think of an ISA as the **software interface for the hardware processor** device. Each **instruction** is a **procedure** provided by the device.

Compilers and ISAs

When a compiler **compiles** a program, it **converts** your **program** into **opcodes** written in the **target ISA**.

```
#include <stdio.h>

int main() {
  printf("Hello world!\n");
  return 0;
}
```

```
e9 2d 48 00
e2 8d b0 04
e5 9f 00 0c
eb ff ff fe
e3 a0 30 00
e1 a0 00 03
e8 bd 88 00
00 00 00 00
```

The resulting file, which is filled with **binary representations of opcodes** (i.e., **machine language**) is usually referred to as a "**binary**."

Instruction Mnemonics

Opcodes are **difficult to understand**. When understanding is important, we use shorthand **labels** called **instruction mnemonics**.

```
e9 2d 48 00
e2 8d b0 04
e5 9f 00 0c
eb ff ff fe
e3 a0 30 00
e1 a0 00 03
e8 bd 88 00
00 00 00 00
```

```
push{fp, lr}
add fp, sp, #4
ldr r0, [pc, #12]
bl  0 <puts>
mov r3, #0
mov r0, r3
pop {fp, pc}
andeq   r0, r0, r0
```

There is a **1:1 correspondence** between **opcodes** and **mnemonics**.

Mnemonic Syntax

You might have seen assembly before, and if so, you probably saw either **AT&T** syntax or **Intel** syntax.

```
lea    0x4(%esp),%ecx
and    $0xfffffff0,%esp
pushl  -0x4(%ecx)
push   %ebp
mov    %esp,%ebp
push   %ecx
sub    $0x4,%esp
sub    $0xc,%esp
push   $0x0
call   1a <main+0x1a>
add    $0x10,%esp
mov    $0x0,%eax
mov    -0x4(%ebp),%ecx
leave
lea    -0x4(%ecx),%esp
ret
```

AT&T

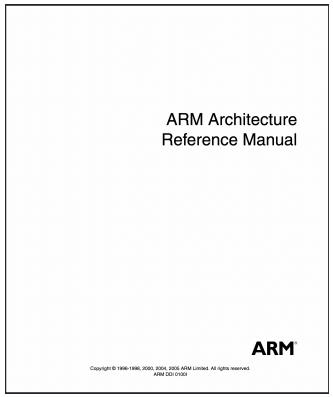
```
lea    ecx,[esp+0x4]
and    esp,0xfffffff0
push   DWORD PTR [ecx-0x4]
push   ebp
mov    ebp,esp
push   ecx
sub    esp,0x4
sub    esp,0xc
push   0x0
call   1a <main+0x1a>
add    esp,0x10
mov    eax,0x0
mov    ecx,DWORD PTR [ebp-0x4]
leave
lea    esp,[ecx-0x4]
ret
```

Intel

ARM has its own syntax! We're using **unified ARM** syntax. It looks a bit like Intel syntax.

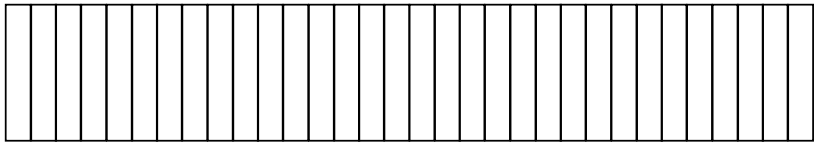
Mnemonic Syntax

Do I remember ARM mnemonics? **Not really**.
I look them up.



ARMv6

"32-bit" refers both to the **size of a basic data unit**, or **word**, for **integers** used in a processor as well as the size of **instructions**.



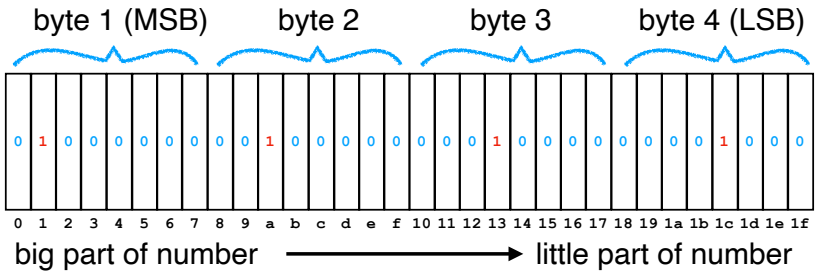
Each cell in the image above stores one **bit** (**binary digit**).

Endianness

Suppose you have the decimal number `1075843080` **stored as a binary number** (as an `unsigned int`).

There are **many ways** to store this number.

The most intuitive format is "**big endian**," where the **most significant bytes are stored first** (before less significant bytes) in memory.



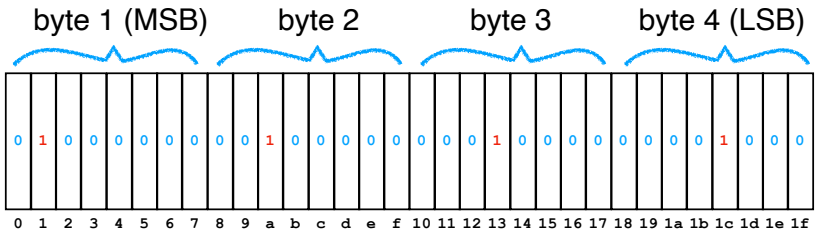
Endianness

ARM processors have **configurable** endianness.

In this class, we will use "**little endian**" format. This means that the **most significant byte is stored last**.

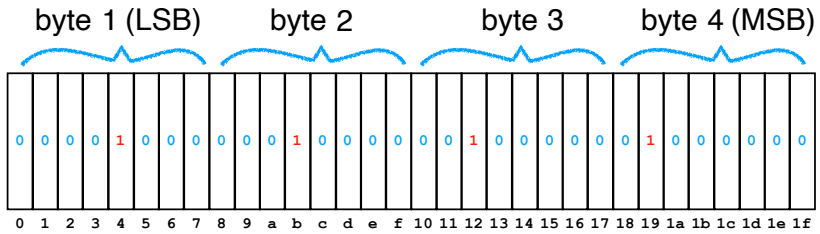
Endianness

Big endian:



Endianness

Little endian:

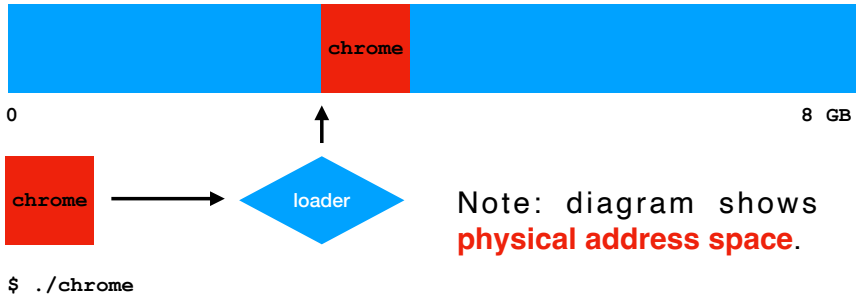


To be clear, this is the **same** decimal number `1075843080` stored in binary. We simply interpret it differently.

Rule: least significant bit is stored at smallest index.

Running a program

The details of how a program is loaded into memory varies by **architecture**, **operating system**, and **language**.



Note: diagram shows **physical address space**.

```
$ ./chrome
```

In Linux, a program called the **loader** reads the program from disk and puts it in memory.

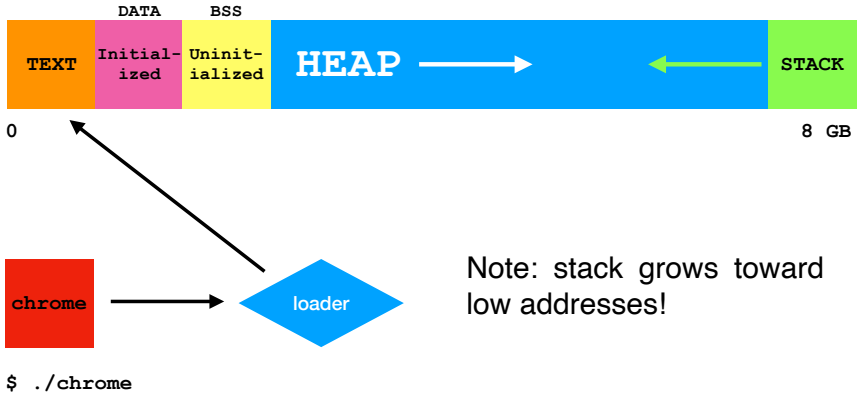
Running a program

After loading the program, on **Linux**, the loader:

1. allocates memory for the runtime call stack,
2. copies CLI program arguments into the stack,
3. calls `_start()`, which starts the C runtime.
4. `_start()` eventually calls `main()`.

Running a program

In the **virtual address space** of the program (e.g., `chrome`), the loader puts

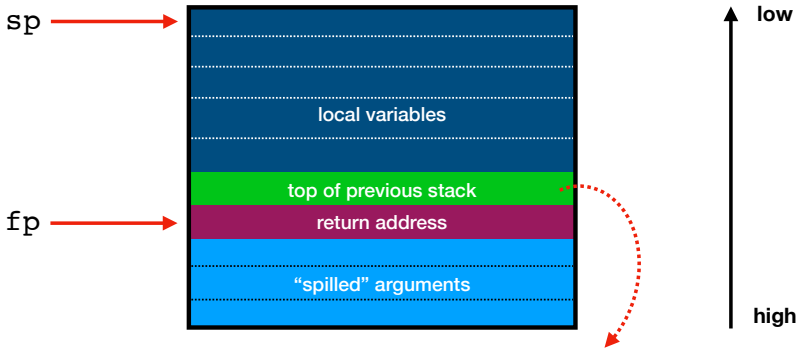


Note: stack grows toward low addresses!

Runtime call stack

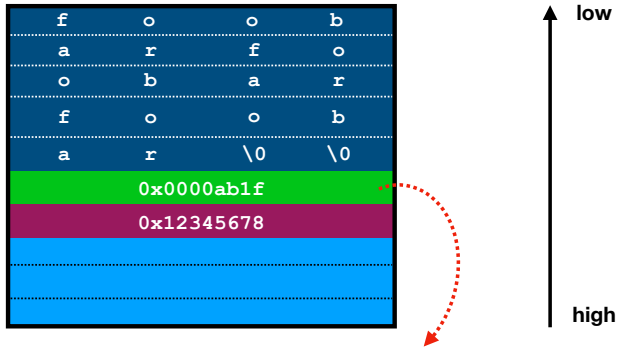
The runtime call stack tracks the **state** of the **currently running function**.

The basic element is a data structure called a **stack frame**.



Stack smashing

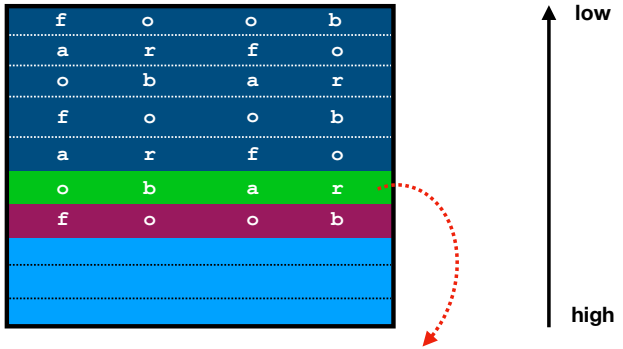
Stack smashing takes advantage of the fact that **writing off the end of a stack-allocated buffer writes toward the return address**.



Stack smashing

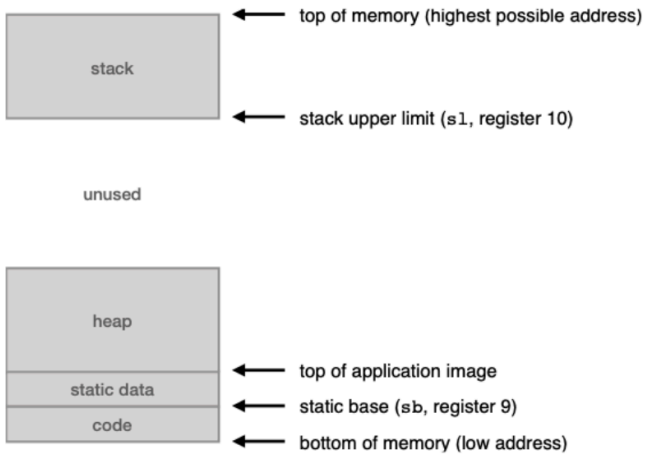
Stack smashing takes advantage of the fact that **writing off the end of a stack-allocated buffer writes toward the return address**.

Program "returns" wherever this points!

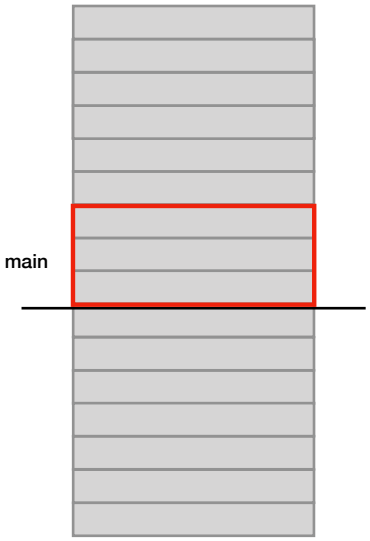


How does a function “happen”?

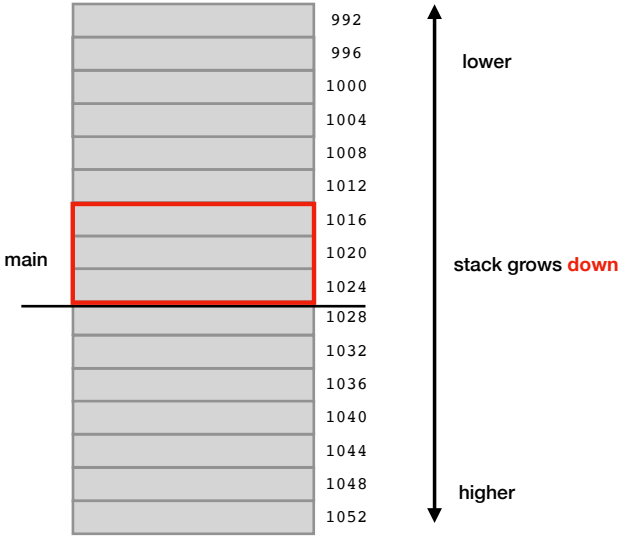
How does a function “happen”?



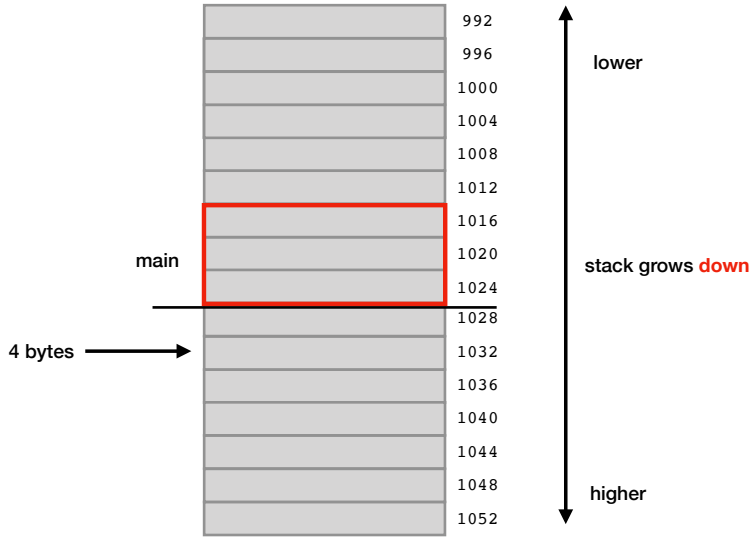
How does a function “happen”?



How does a function “happen”?



How does a function “happen”?



Calling convention

A **calling convention** is a **specification** for the functioning of a call stack. Calling conventions describe:

- **How** parameters are passed to a function.
- **The order** in which parameters are passed.
- **Which registers** are used to store stack metadata.
- **Who saves** registers (**caller** or **callee**), and
- **Who restores** registers (**caller** or **callee**) after calling.

This information is **necessary** to ensure that code generated by different compilers **interoperates**.

ARM Calling Convention

How functions “work” for the C language on 32-bit ARM machines running UNIX.

- **How** parameters are passed to a function.
  ✓**in registers; spill to the stack**
- **The order** in which parameters are passed.
  ✓**right-to-left**
- **Which registers** are used to store stack metadata.
  ✓**`pc`: program counter (i.e., instruction pointer)**
  ✓**`sp`: pointer to top of stack**
  ✓**`fp`: pointer to bottom of stack**
- **Who saves** registers,
  ✓**callee saves `v1-v5`, `fp`, `sp`, etc; caller saves `lr`.**
- **Who restores** registers after calling.
  ✓**callee restores `v1-v5`, `fp`, `sp`, etc.; callee restores lr**

This program does *almost* nothing.

```
void foo() {}

int main() {
  foo();
}
```

What does it do?

Recap & Next Class

Today we learned:

How C functions work

Next class:

How argument passing works