

044101

Lecture #1

Goals

- Understand how **computers actually work** from the software perspective
- Hands on: learn and practice **Systems Programming** basics
- Useful outcomes:
 - Understand hardware and software interact
 - Become more effective programmer
 - Prepare for later courses: OS, Compilers, Networking, Computer Structure, Computer Security

Your **first zipper course** in Computer Engineering

It's important to understand how things really work!

- All you have learned so far are **software abstractions**
 - File, Memory, Network

They hide how things work to make it easier to program them

- But!
 - To debug you'll need to understand what **may go wrong**
 - Sometimes you'll have to break abstractions to get **better performance**
 - Security bugs are **always beneath the surface**

(*בנוסף לשליטה על הsurface יש לזכור שפה של מילויים מתחום הsurface*)

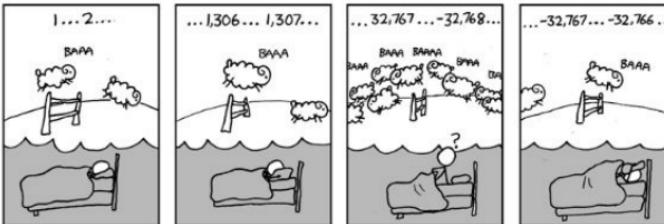
Examples when abstractions might be misleading..

Great Reality #1:

Ints are not Integers, Floats are not Reals

■ Example 1: Is $x^2 \geq 0$?

- Float's: Yes!



- Int's:

- $40000 * 40000 \rightarrow 16000000000$
- $50000 * 50000 \rightarrow ?$

(Overflow)

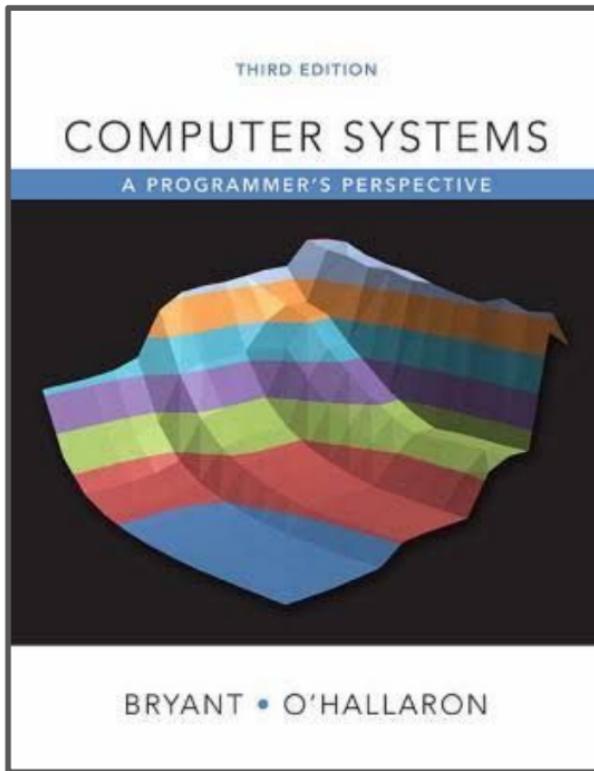
■ Example 2: Is $(x + y) + z = x + (y + z)$?

- Unsigned & Signed Int's: Yes!

- Float's:

- $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
- $1e20 + (-1e20 + 3.14) \rightarrow ??$

Textbook



(-UFECN -UFGM 9 - 100)
gfrir >/kr >/023

Online materials, including videos from the authors:

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f18/www/schedule.html>

We do not teach everything, and not everything is in the book

But the book is great and extremely useful!

Home assignments

(*הסניפיםicos, אולי שאלות מילויים*)

- 30% taken
- Almost once a week
- Most are short (2-3 hours of intensive work)
- Few wet, few dry
- **Allowed to miss ONE except for the last to get full grade**
- Goals:
 - Involvement during the semester, not at the exam
 - This course is NOT suitable for binge-watching
 - Simple, but insightful
 - You can't learn swimming without getting wet

They are for you! Plagiarism will be detected and punished!

How computers run programs (Book: Chapter 1)

Hello World

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

Source code - human
readable

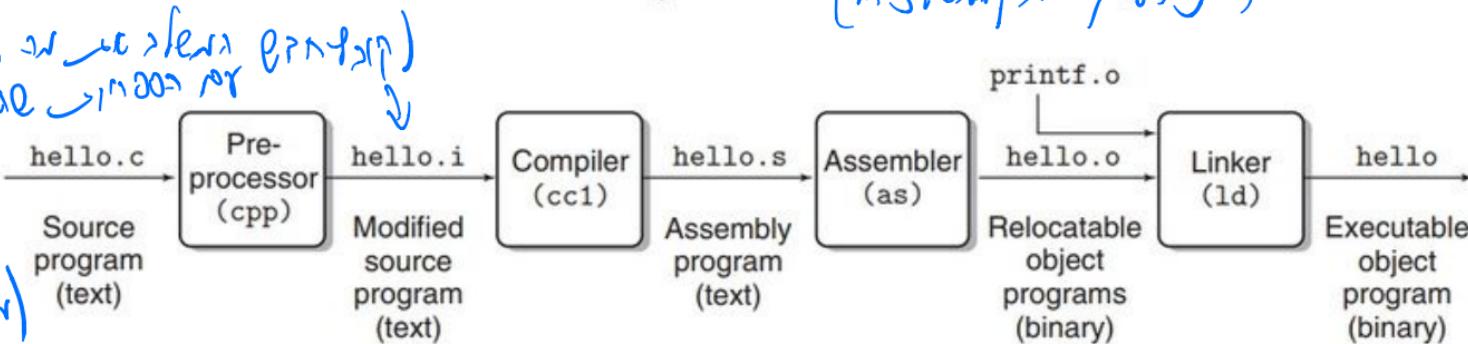
Saved as in file

Encoded “hello world” in ASCII

#	i	n	c	l	u	d	e	SP	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	SP	m	a	i	n	()	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	SP	SP	SP	SP	p	r	i	n	t	f	("	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	SP	w	o	r	l	d	\	n	")	;	\n	SP
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	SP	SP	r	e	t	u	r	n	SP	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

Compilation: source-code → machine language

(assembly code)



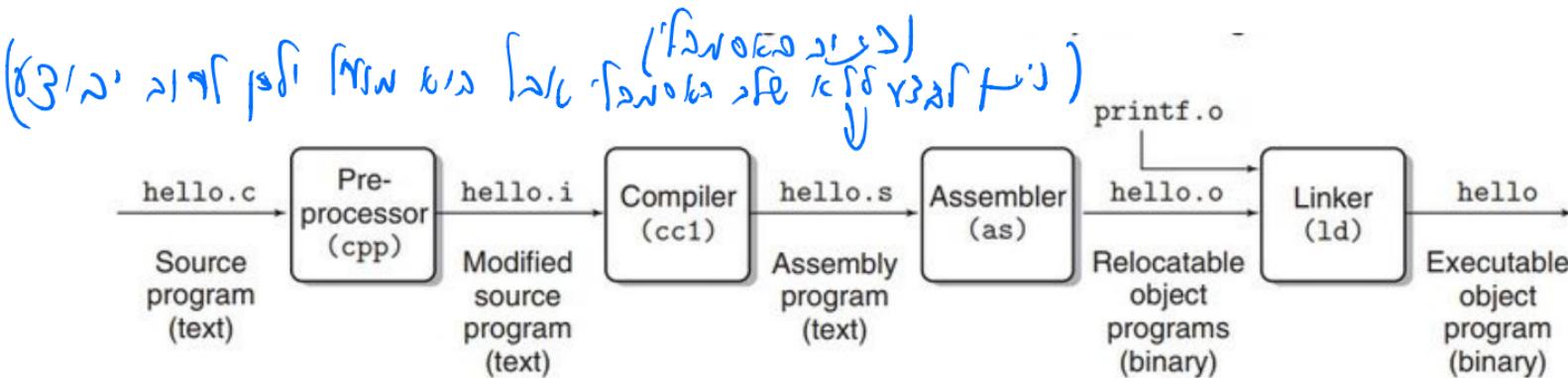
(use macro definitions)
(join symbols)

(input)
(new code)
(preProc)
1
2

```
#include <stdio.h>

3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

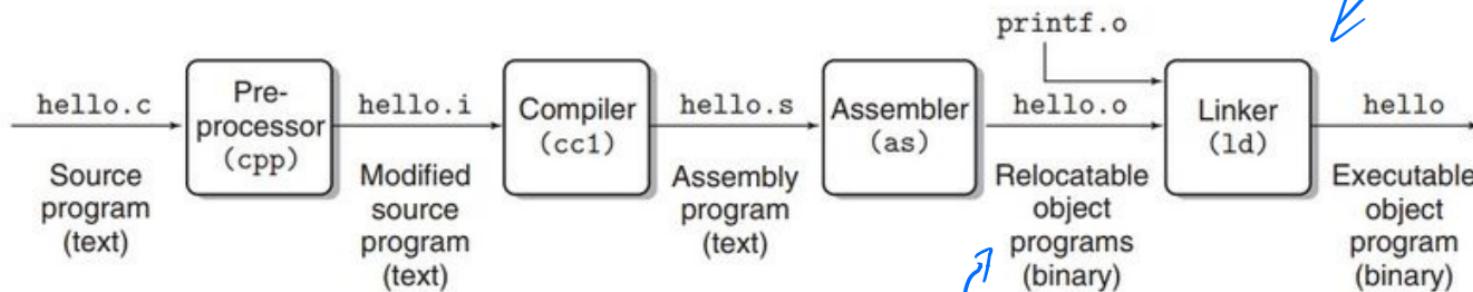
Compilation: source-code -> machine language



```
1 main:  
2     subq    $8, %rsp  
3     movl    $.LC0, %edi  
4     call    puts  
5     movl    $0, %eax  
6     addq    $8, %rsp  
7     ret
```

Compilation: source-code -> machine language

(היפר פайл הינו (הה רג'ן פון) ".o" - קבץ אובייקט מילויים)



0000000000000000 <main>:
0: 55
1: 48 89 e5
4: 48 8d 3d 00 00 00 00
b: e8 00 00 00 00 00
10: b8 00 00 00 00 00
15: 5d
16: c3

Poll: linker/compiler

Yes/No

האם file.exe מודפס כטקסט ASCII (ASCII בדרכו). C פורט נסמן (כן/לא)

- Compiler translates text into ASCII

- Linker resolves code locations unknown at compilation time

- Compiler operates one-file-at-a-time

- A compiler produces an assembly code in ASCII

Great Reality #2: You've Got to Know Assembly

- Chances are, you'll never write programs in assembly
 - Compilers are much better & more patient than you are
- But: Understanding assembly is key to machine-level execution model
 - Behavior of programs in presence of bugs
 - High-level language models break down
 - Tuning program performance
 - Understand optimizations done / not done by the compiler
 - Understanding sources of program inefficiency
 - Implementing system software
 - Compiler has machine code as target
 - Operating systems must manage process state
 - Creating / fighting malware
 - x86 assembly is the language of choice!

Asking an OS to invoke a program

Command line: program invocation and interaction in **shell**

(Linux - ממשק משתמש, מילויים)

```
linux> ./hello
hello, world
linux>
```

Poll: yes/no

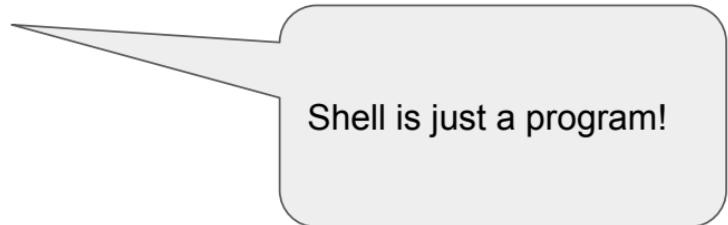
Shell is an operating system ✓

There is no operating system without a Shell ✓ (יש לנוoperating system без shell)

Shell commands limited to a pre-defined set ✗ X

Asking an OS to invoke a program

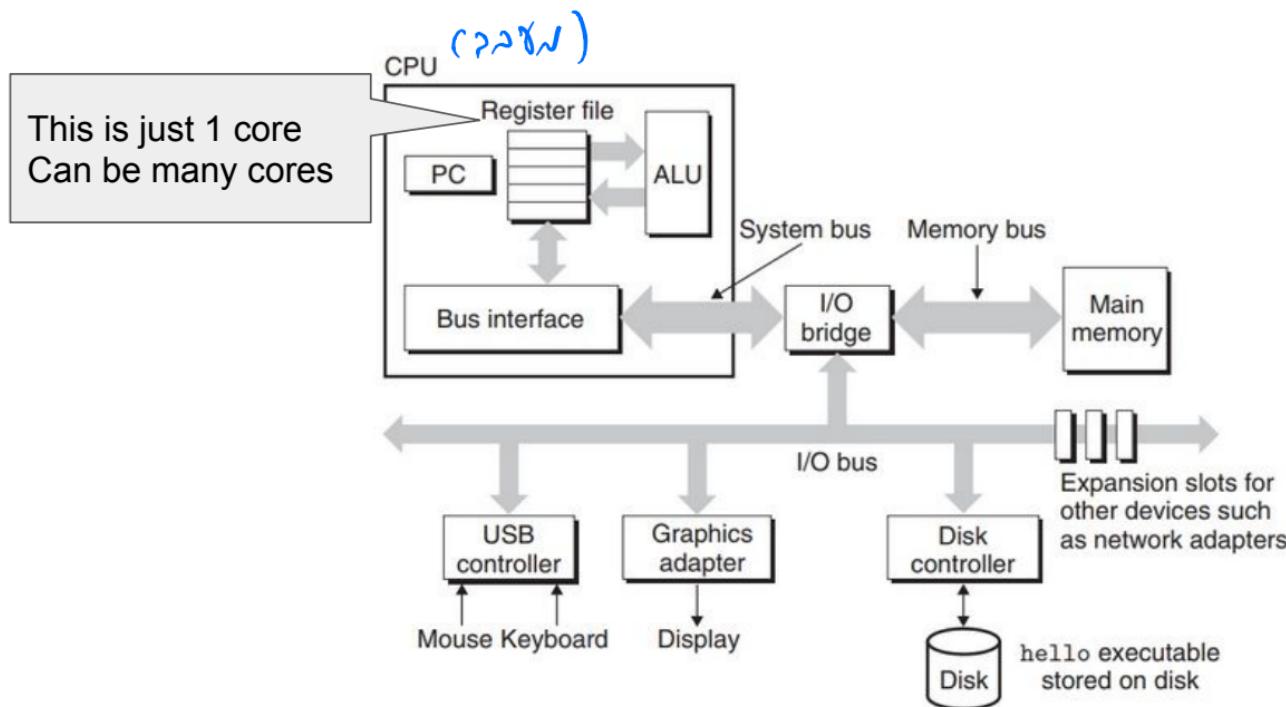
Command line: program invocation in **shell**



Shell is just a program!

```
linux> ./hello  
hello, world  
linux>
```

What's inside the computer



Poll: what do you know about the internals

Yes/No

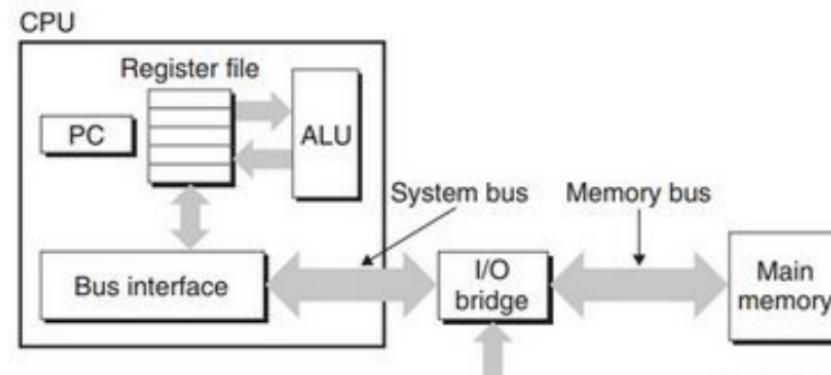
1. Have you ever installed a new HDD/SSD in a computer? *y*
2. Internal bus in a PC can transfer up to 16GB/s *y*

Processor executing a program

(*פונקציית ISA*)

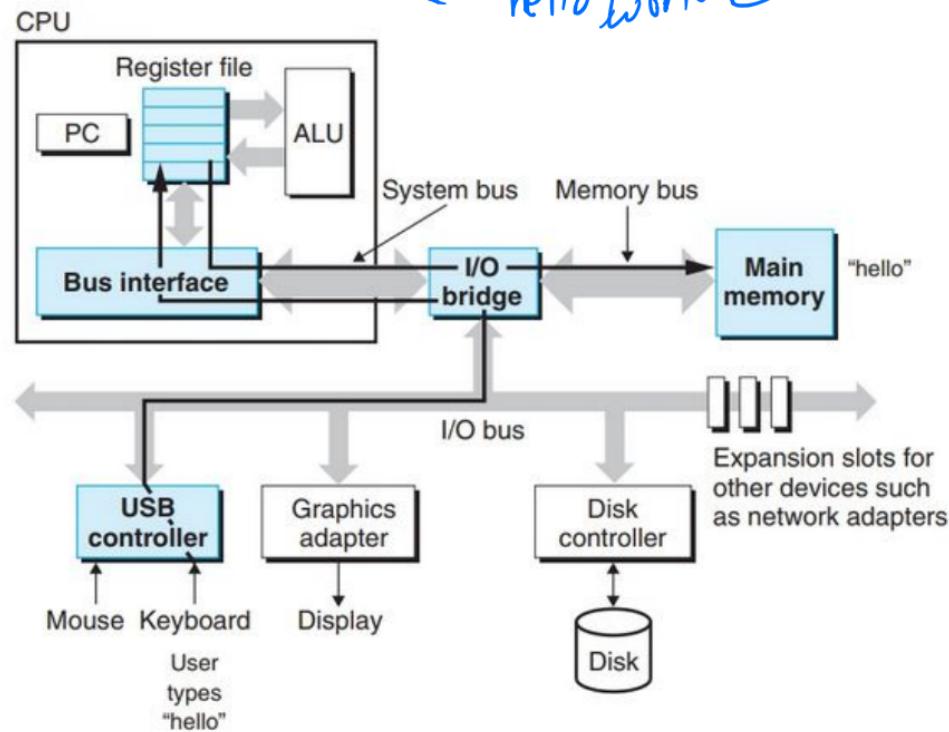
Instruction Set Architecture (ISA): the interface to the processor

ISA includes CPU instructions: load/store/operate/jump



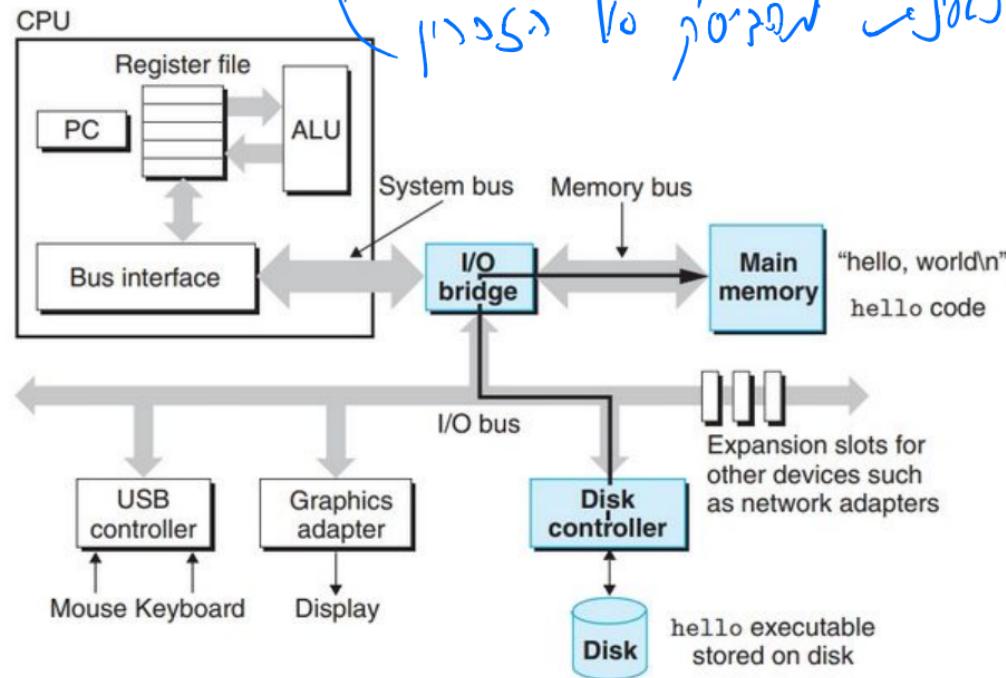
Typing “hello world”

(*பிடிப்புவதற்கிடமிருப்பது*)
hello world *கீல்வதற்கிடமிருப்பது*)



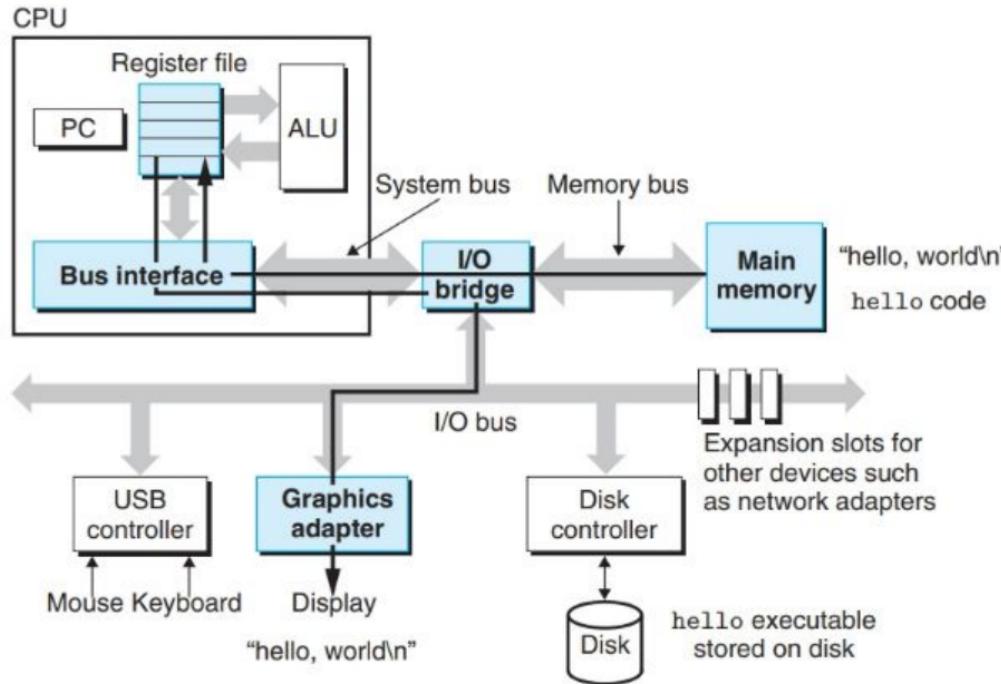
Loading from the disk

(הdisk יSEND לCPU מdisk)

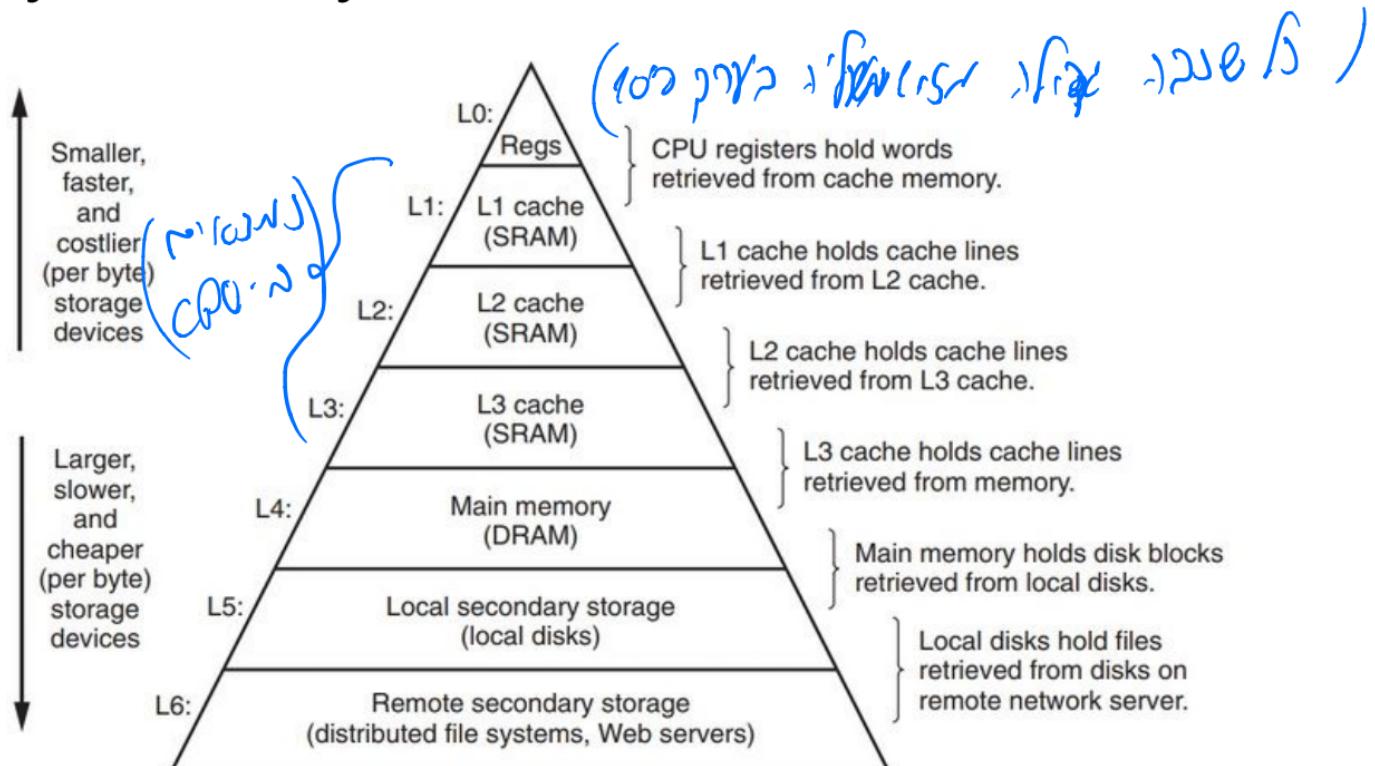


Writing the output

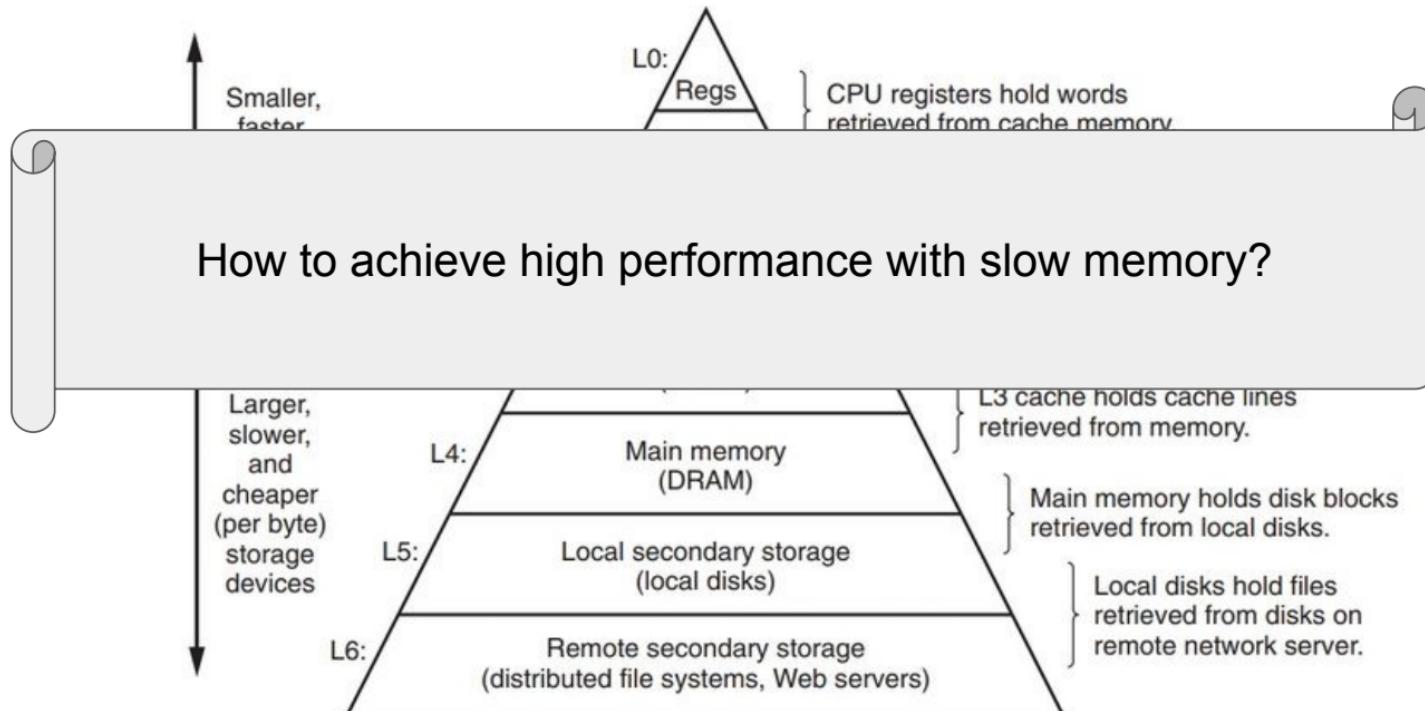
(*הציגו פונקציית כתיבת תוצאות*)



Memory hierarchy



Memory hierarchy



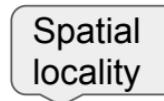
Concept of a cache

Small, close-to-CPU and fast!

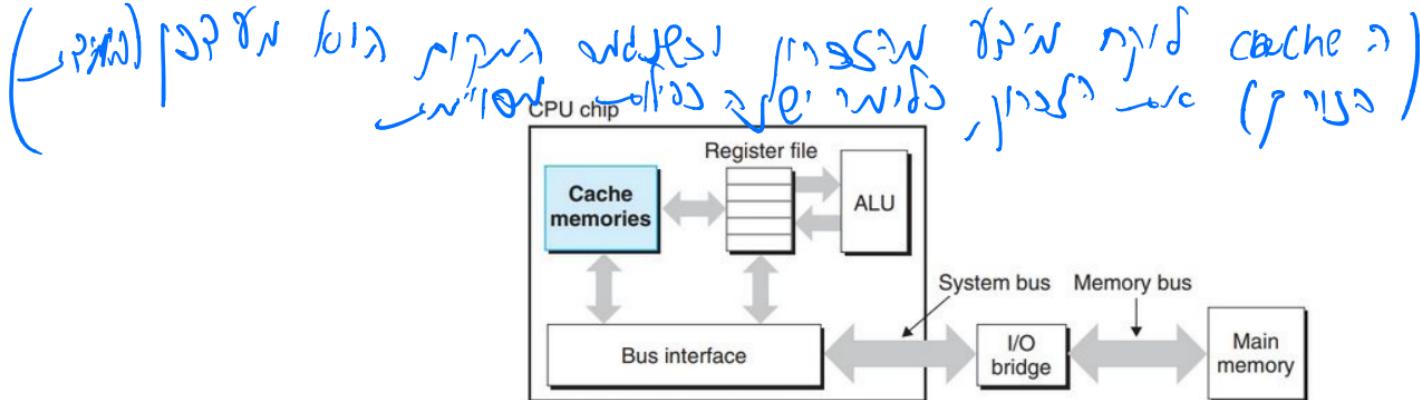
(ρ_{air} , 10^5 Pa)

(Reg-f) \rightarrow $\text{10}^{\text{C}} \downarrow \text{10}^{\text{C}}$ \rightarrow 10^{C} \rightarrow 10^{C} \rightarrow 10^{C}

Exploits locality of accesses:



most of the accesses to the close locations are close in time



Memory System Performance Example

```
void copyij(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (i = 0; i < 2048; i++)  
        for (j = 0; j < 2048; j++)  
            dst[i][j] = src[i][j];  
}
```

```
void copyji(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (j = 0; j < 2048; j++)  
        for (i = 0; i < 2048; i++)  
            dst[i][j] = src[i][j];  
}
```

YES

Poll: which execution is faster

NO

Memory System Performance Example

(גִּבְעָן לְמִלְחָמָה כַּפָּן)

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

4.3ms

2.0 GHz Intel Core i7 Haswell

81.8ms

- Hierarchical memory organization
- Performance depends on access patterns
 - Including how step through multi-dimensional array

Great Reality #4: There's more to performance than asymptotic complexity



- Constant factors matter too!
- And even exact op count does not predict performance
 - Easily see 10:1 performance range depending on how code written
 - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
 - How programs compiled and executed
 - How to measure program performance and identify bottlenecks
 - How to improve performance without destroying code modularity and generality

Summary so far

Programs written in C (text) are translated into binary

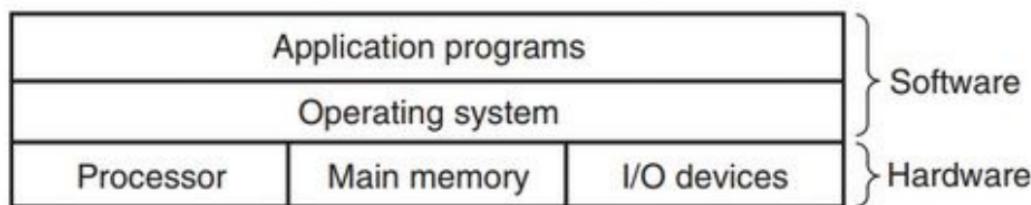
A computer comprises many components, inter-connected via bus

A processor runs a program

Memory is organized in a hierarchy, caching is a common concept essential for achieving system performance

OS hardware management

Operating System is a management layer to allow easy and secure use of hardware by exposing **abstractions**



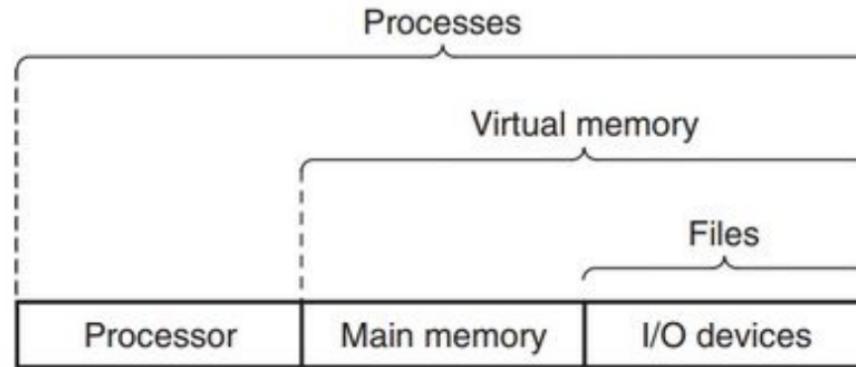
(הארχיטקטורה היא מבנה של איסטרטגיית
המערכת ותפקידים ורשותם בפעולתה)

(ה'אליך גודל ועוצמה נוראה) סדרת מילויים בפיזיקה וביולוגיה

There are hundreds of processes running on a machine

All processes are isolated

Each has its own **virtual (almost unbounded) memory space**



Great Reality #3: Memory Matters

Random Access Memory Is an Unphysical Abstraction

(*जब तक कि हमें यह समझ नहीं लगता कि मैं इनमें से किसी भी विषय को जानता हूँ।*)

■ Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated

■ Memory referencing bugs especially pernicious

- Effects are distant in both time and space

■ Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

```
fun(0) --> 3.14
fun(1) --> 3.14
fun(2) --> 3.1399998664856
fun(3) --> 2.00000061035156
fun(4) --> 3.14
fun(6) --> Segmentation fault
```

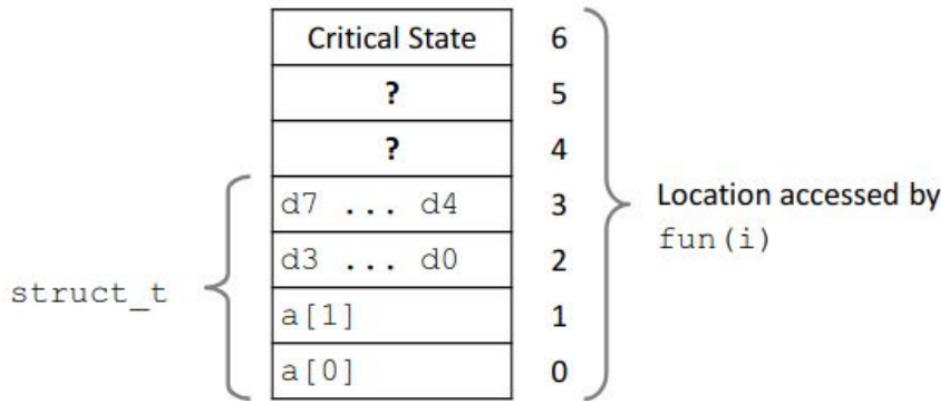
- Result is system specific

Memory Referencing Bug Example

```
typedef struct {  
    int a[2];  
    double d;  
} struct_t;
```

fun(0) -->	3.14
fun(1) -->	3.14
fun(2) -->	3.1399998664856
fun(3) -->	2.00000061035156
fun(4) -->	3.14
fun(6) -->	Segmentation fault

Explanation:



Poll: malloc vs. free

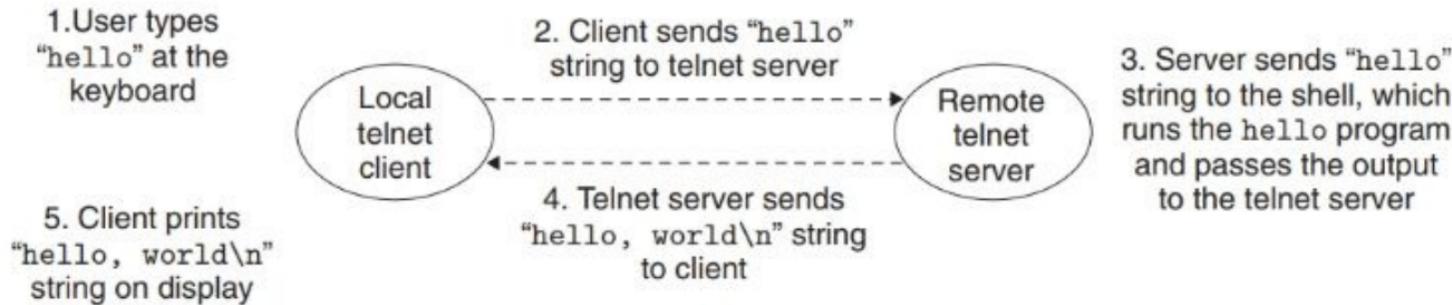
Yes/No

1. The example does not make sense
2. Access to unallocated memory causes segfault *(no! no!)*
3. If I did malloc for 4KB I can free 2KB and then 2KB more
4. I must free a static array declared as a local variable in a function

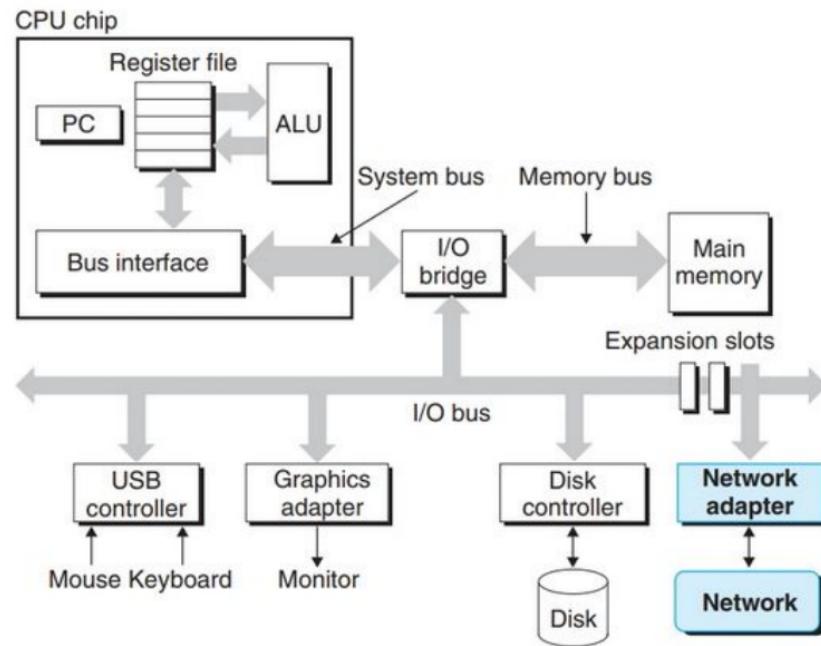
Memory Referencing Errors

- **C and C++ do not provide any memory protection**
 - Out of bounds array references
 - Invalid pointer values
 - Abuses of malloc/free
- **Can lead to nasty bugs**
 - Whether or not bug has any effect depends on system and compiler
 - Action at a distance
 - Corrupted object logically unrelated to one being accessed
 - Effect of bug may be first observed long after it is generated
- **How can I deal with this?**
 - Program in Java, Ruby, Python, ML, ...
 - Understand what possible interactions may occur
 - Use or develop tools to detect referencing errors (e.g. Valgrind)

Networking



Networking - how it works



Great Reality #5: Computers do more than execute programs

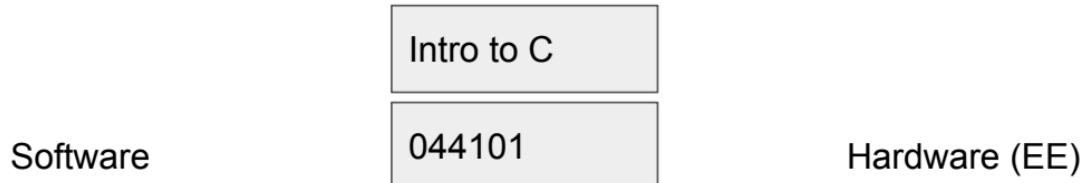
- **They need to get data in and out**
 - I/O system critical to program reliability and performance

- **They communicate with each other over networks**
 - Many system-level issues arise in presence of network
 - Concurrent operations by autonomous processes
 - Coping with unreliable media
 - Cross platform compatibility
 - Complex performance issues

Our course is programmer-centric

- Our Course is Programmer-Centric
 - By knowing more about the underlying system, you can be more effective as a programmer
 - Enable you to write programs that are more reliable and efficient
 - Incorporate features that require hooks into OS
 - Cover material in this course that you won't see elsewhere
 - Not just a course for dedicated hackers
 - We bring out the hidden hacker in everyone!

Role of 044101 in Computer Engineering Curriculum



Intro to OS (CS/EE)
OS design and implementation (CS)
Compilers (EE)
Binary Translation (EE)
Computer networks (CS/EE)
Intro and tools in cyber security (EE)
Network security (CS)
Reverse engineering (CS)
Fundamentals of Distributed systems (EE)
Distributed and parallel programming (CS)
Object Oriented Design (CS/EE)
Functional Distributed Programming (EE)
Accelerators and accelerated systems (CS/EE)

Intro to computer architectures
Microprocessors
VLSI
Memristors
Ultra-fast networking
Parallel computer architecture
Advanced computer architecture

Logistics

1. Lectures - recordings will be made available online. But do attend them!
2. Book is a great resource
3. Online lectures by the book authors:
<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f18/www/schedule.html>
4. Workshops - optional, but desirable
5. Tutorials help practice the lecture materials
6. Do your home assignments - **they are FOR YOU!**
7. Reception hours - upon request

044101

Lecture #2

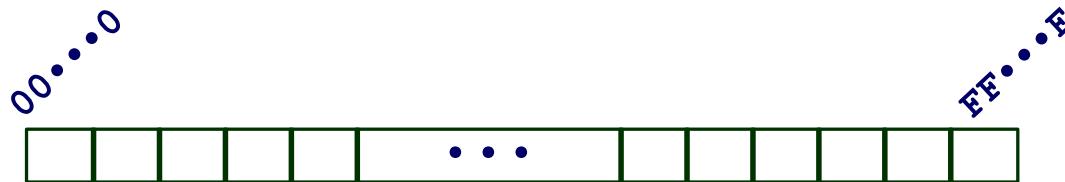
Data representation (CS:App: Chapter 2)

Ints

Bits

Endianness

Byte-Oriented Memory Organization



(memory is just a big sequential store of processes in memory pages)

■ Programs refer to data by address

- Conceptually, envision it as a very large array of bytes
 - In reality, it's not, but can think of it that way
- An address is like an index into that array
 - and, a pointer variable stores an address

■ Note: system provides private address spaces to each “process”

- Think of a process as a program being executed
- So, a program can clobber its own data, but not that of others

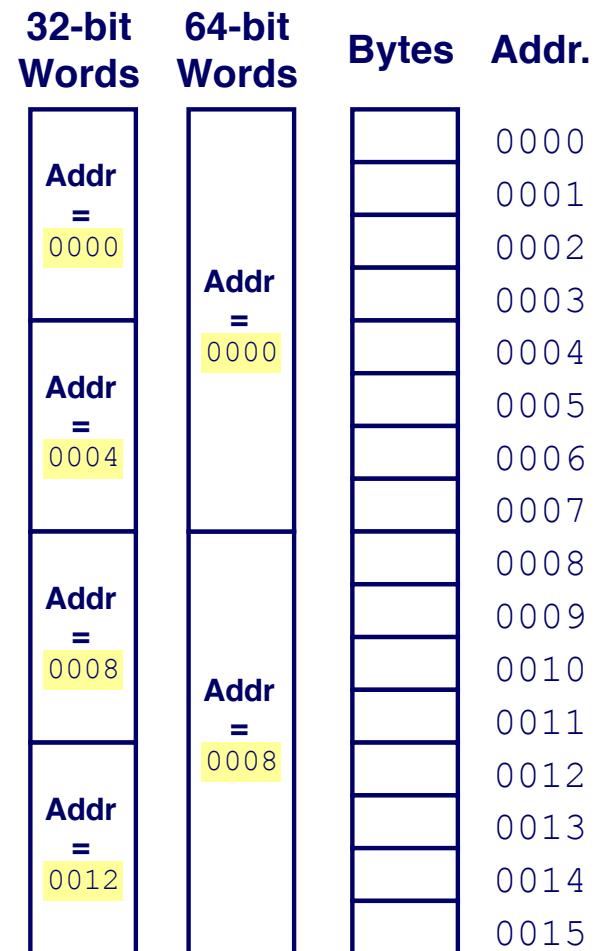
Machine Words

■ Any given computer has a “Word Size”

- Nominal size of integer-valued data
 - and of addresses
- Until recently, most machines used 32 bits (4 bytes) as word size
 - Limits addresses to 4GB (2^{32} bytes)
- Increasingly, machines have 64-bit word size
 - Potentially, could have 18 EB (exabytes) of addressable memory
 - That's 18.4×10^{18} 18 exabytes
- Machines still support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
 - Address of first byte in word
 - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Example Data Representations

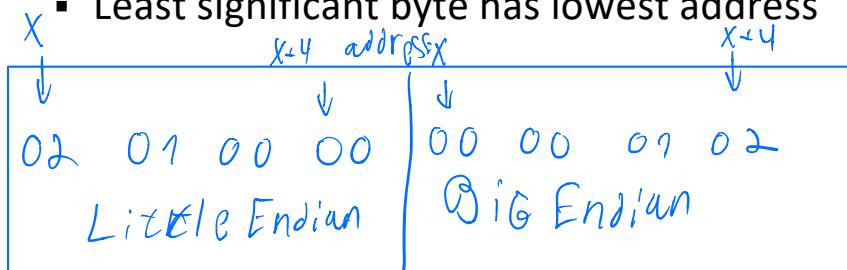
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

Byte Ordering

عکس این جهت را می‌دانیم ۱۷۰
ماشینی ۱۷۰

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions

- Big Endian: Sun (Oracle SPARC), PPC Mac, *Internet*
 - Least significant byte has highest address
- Little Endian: *x86*, ARM processors running Android, iOS, and Linux
 - Least significant byte has lowest address

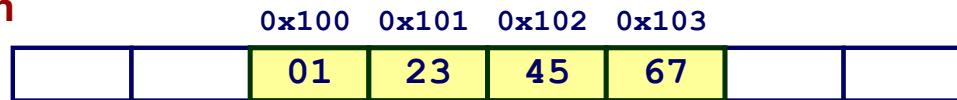


Byte Ordering Example

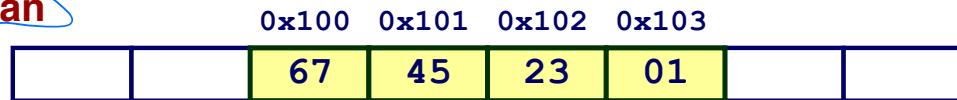
■ Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

Big Endian



Little Endian



Representing Integers

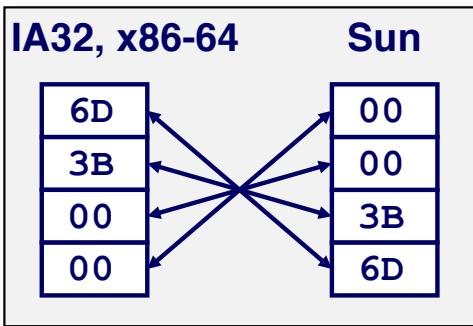
Decimal: 15213

Binary: 0011 1011 0110 1101

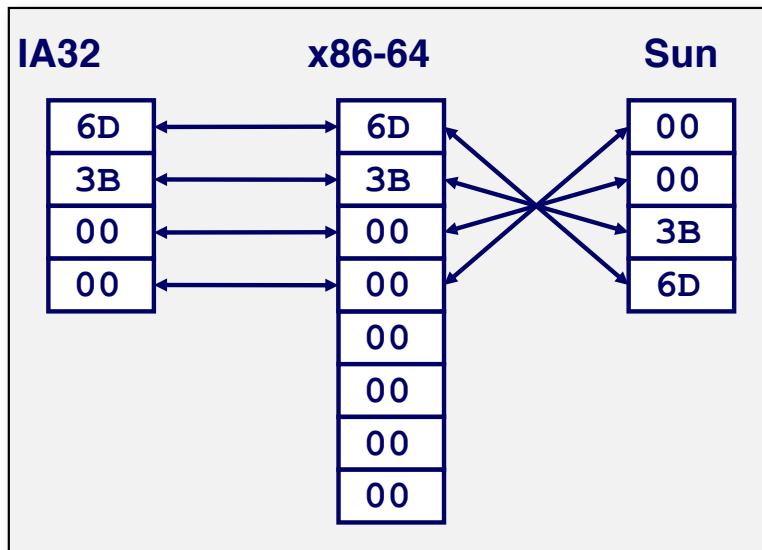
Hex: 3 B 6 D

`int A = 15213;`

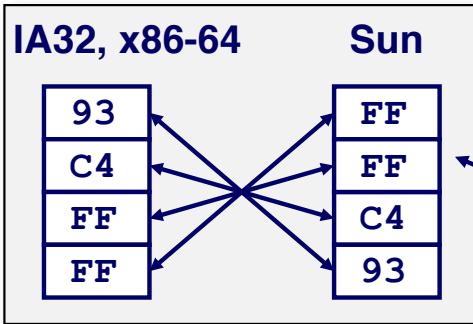
Increasing addresses ↓



`long int C = 15213;`



`int B = -15213;`



Two's complement representation

Examining Data Representations

- Code to Print Byte Representation of Data
 - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:

%p: Print pointer
%x: Print Hexadecimal

show_bytes Execution Example

```
int a = 15213;  
printf("int a = 15213;\n");  
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

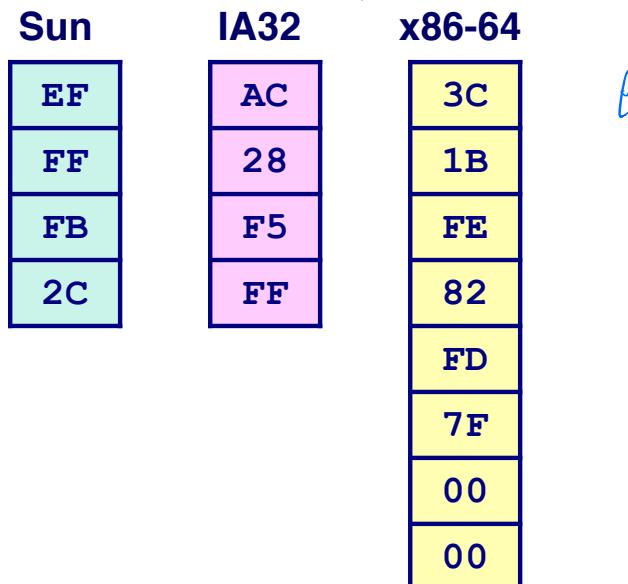
	int a = 15213;	
X	0x7ffffb7f71dbc	6d <i>LSB</i>
	0x7ffffb7f71dbd	3b
	0x7ffffb7f71dbe	00
X-4	0x7ffffb7f71dbf	00 <i>MSB</i>

Little Endian of 7f71 dbf0 003b 6d

Representing Pointers

```
int B = -15213;  
int *P = &B;
```

(00 00 7F FD 82 FE 1B 3C) +
=> (00 00 7F FD 82 FE 1B 3D)



Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

```
char S[6] = "18213";
```

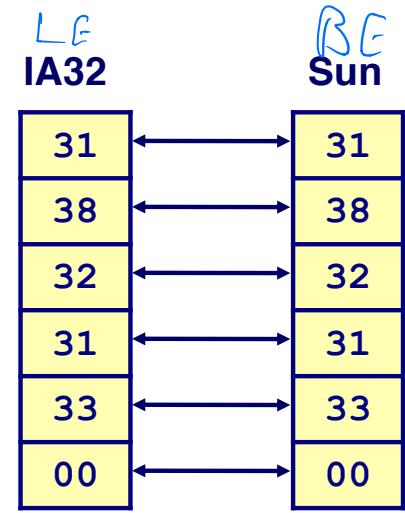
■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
 - Standard 7-bit encoding of character set
 - Character “0” has code 0x30 *Short int [0]*
 - Digit i has code $0x30+i$ *Short int [i]*
- String should be null-terminated
 - Final character = 0

■ Compatibility

- Byte ordering not an issue

*Short int [0] = 0x0100
 Short int [1] = 0x0302*



(Little Endian vs Big Endian, 18213, 18213 → 18213)

Reading Byte-Reversed Listings

■ Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code

■ Example Fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0,0x28(%ebx)

■ Deciphering Numbers

- Value:
- Pad to 32 bits:
- Split into bytes:
- Reverse:

The diagram illustrates the process of deciphering the number 0x12ab. It starts with the byte-reversed assembly instruction `add $0x12ab,%ebx`. A red arrow points from the byte `ab` in the instruction code to the value `0x12ab`. Another red arrow points from the byte `ab` to the value `0x000012ab`, which is the number padded to 32 bits. Below that, the bytes are split into individual bytes: `00 00 12 ab`. Finally, the bytes are reversed to their original order: `ab 12 00 00`.

0x12ab
0x000012ab
00 00 12 ab
ab 12 00 00

Reminder 1

- Byte = 8 bits
- Kilo=2^E10, Mega=2^E20, Giga=2^E30, Tera=2^E40, Peta=2^E50, Exa=2^E60
- Boolean algebra: XOR(^), AND(&), OR(|), NOT(~)

Poll: find the mistake:

Yes-1

No-2

Go sower -3

Go faster - 4

$$\begin{array}{l} 1: 100 \& 101 = 100 \\ 2: \sim 11101 = 10 \\ 3: 110 | 100 = 110 \\ 4: 111 ^ 101 = 1 \end{array}$$

↳ 100

(bitwise is 0) 010

Reminder 1

- Byte = 8 bits
- Kilo=2E10, Mega=2E20, Giga=2E30, Tera=2E40, Peta=2E50, Exa=2E60
- Boolean algebra: XOR(^), AND(&), OR(|), NOT(~)

Logical

Mind the difference in C!

	is not		Bitwise
&&	is not	&	
!	is not	~	

```
if (0b10 & 0b101){  
    printf("bitwise")  
}
```

```
if (0b10 && 0b101){  
    printf("logic")  
}
```

Example: Representing & Manipulating Sets

■ Representation

- Width w bit vector represents subsets of $\{0, \dots, w-1\}$
- $a_j = 1$ if $j \in A$

Width bit vector of size w contains 1 at index j if $j \in A$

- 01101001 $\{0, 3, 5, 6\}$
- 76543210

- 01010101 $\{0, 2, 4, 6\}$
- 76543210

■ Operations

- & Intersection 01000001 $\{0, 6\}$
- | Union 01111101 $\{0, 2, 3, 4, 5, 6\}$
- ^ Symmetric difference 00111100 $\{2, 3, 4, 5\}$
- ~ Complement 10101010 $\{1, 3, 5, 7\}$

Reminder 2

223 = decimal

0b11011111 = binary

0337 = octal

0xBF = hexadecimal

0b1101001 into octal?

0b100**1**001 into hex?

v q

Useful constants

0b11111111 = 0xFF =255

0b1 = 0x1 = 01=1

0x1000 = 4KB

0b1010 = 10 = 0xA

0b1111 = 15 = 0xF

Shift Operations

■ Left Shift: $x \ll y$

- Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right

■ Right Shift: $x \gg y$

- Shift bit-vector x right y positions
 - Throw away extra bits on right
- Logical shift
 - Fill with 0's on left
- Arithmetic shift
 - Replicate most significant bit on left

■ Undefined Behavior

- Shift amount < 0 or \geq word size

↳ This is undefined Endian

Give the JB b'

Shift Operations

Arith. & Log. Shifts

■ Left Shift: $x \ll y$

- Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right

■ Right Shift: $x \gg y$

- Shift bit-vector x right y positions
 - Throw away extra bits on right
- Logical shift
 - Fill with 0's on left
- Arithmetic shift
 - Replicate most significant bit on left

■ Undefined Behavior

- Shift amount < 0 or \geq word size

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Sign \Rightarrow VLSIC
Jew

Data types

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

Size in bytes

Word size in bytes

What is the maximum size of memory addressable with 64bit vs. 32bit word?

Integers

- Signed/unsigned
- Conversion/casting
- Expanding/truncating
- Addition

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;  
short int y = -15213;
```

Sign Bit

- C does not mandate using two's complement

- But, most machines do, and we will assume so

- C short 2 bytes long

$$y = -x + 1$$

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

$$y - x = 0$$

- Sign Bit

- For 2's complement, most significant bit indicates sign
 - 0 for nonnegative
 - 1 for negative

Two-complement: Simple Example

$$\begin{array}{r} \text{-16} & 8 & 4 & 2 & 1 \\ 10 = & 0 & 1 & 0 & 1 & 0 \end{array} \quad 8+2 = 10$$

$$\begin{array}{r} \text{-16} & 8 & 4 & 2 & 1 \\ -10 = & 1 & 0 & 1 & 1 & 0 \end{array} \quad -16+4+2 = -10$$

Numeric Ranges

■ Unsigned Values

- $UMin = 0$
000...0
- $UMax = 2^w - 1$
111...1

■ Two's Complement Values

- $TMin = -2^{w-1}$
100...0
- $TMax = 2^{w-1} - 1$
011...1
- Minus 1
111...1

Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

(in python this would result in overflow)

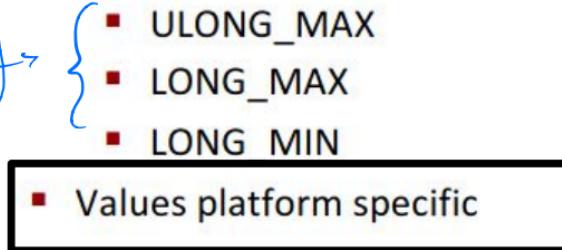
Observations

- $|TMin| = TMax + 1$
 - Asymmetric range
- $UMax = 2 * TMax + 1$

(溢出会导致什么)

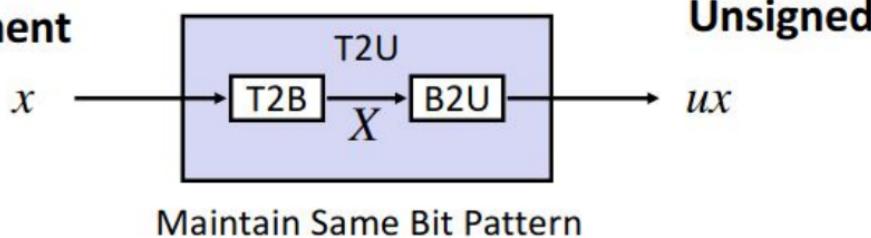
C Programming

- #include <limits.h>
- Declares constants, e.g.,
 - ULONG_MAX
 - LONG_MAX
 - LONG_MIN
- Values platform specific



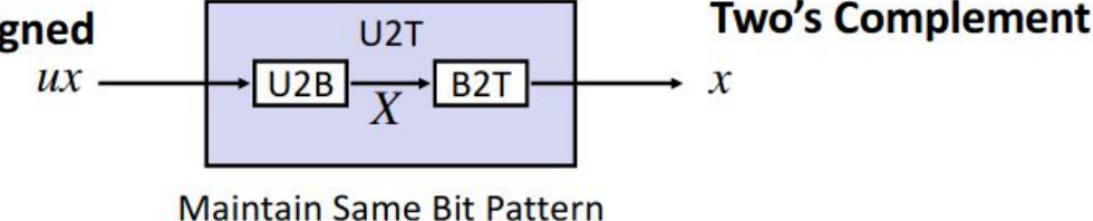
Mapping Between Signed & Unsigned

Two's Complement



Unsigned

Unsigned



Two's Complement

- Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

Mapping Signed ↔ Unsigned



Signed vs. Unsigned in C

■ Constants

- By default are considered to be signed integers
 - Unsigned if have “U” as suffix

0U, 4294967259U

■ Casting

- Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux;  
uy = (unsigned) ty;
```

- Implicit casting also occurs via assignments and procedure calls

tx = ux;
uy = ty;

```
int fun(unsigned u);  
uy = fun(tx);
```

Technion EE M

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Poll (yes/no)

1. `int a= 0xFFFFFFF. (a+1) produces different results for 32bit and 64bit CPU` (111 1111 1111 1111 1111 1111 1111 1111) n /
2. `short int a= 0xFFFF; (1111 1111 1111 1111) -1 + 1` px. plz help me understand?)
`short int b=(short int)((unsigned short int) a);`
`a is not equal b` n ↴
3. `char a,b,c; a=0xFF; b=~a; c=a+1; b is equal c` y ✓
`b = ~a` (1111 1111 1111 1111) 1111 1111 1111 1111

Casting Surprises

Si -> U fl unsigned if l: 'C to ASCII p/c
U -> l ASCII ST -> B SC

■ Expression Evaluation

- If there is a mix of unsigned and signed in single expression,
signed values implicitly cast to unsigned
- Including comparison operations $<$, $>$, $==$, $<=$, $>=$
- Examples for $W = 32$: **TMIN = -2,147,483,648** , **TMAX = 2,147,483,647**

■ Constant₁ Constant₂ Relation Evaluation

0	0U	=	unsigned
-1	0	<	signed

2147483647	-2147483647-1	>	signed
------------	---------------	---	--------

-1	-2	>	signed
----	----	---	--------

2147483647	2147483648U	<	unsigned
------------	-------------	---	----------

Casting Surprises

■ Expression Evaluation

- If there is a mix of unsigned and signed in single expression,
signed values implicitly cast to unsigned
- Including comparison operations $<$, $>$, $==$, $<=$, $>=$
- Examples for $W = 32$: **TMIN = -2,147,483,648** , **TMAX = 2,147,483,647**

Constant ₁	Constant ₂	Relation	Evaluation
0	0U	$=$	unsigned
-1	0	$<$	signed
-1	0U	$>$	unsigned
2147483647	-2147483647-1	$>$	signed
2147483647U	-2147483647-1	$<$	unsigned
-1	-2	$>$	signed
(unsigned)-1	-2	$>$	unsigned
2147483647	2147483648U	$<$	unsigned
2147483647	(int) 2147483648U	$>$	signed

uf
(,12)

Summary

Casting Signed \leftrightarrow Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

- Expression containing signed and unsigned int
 - int is cast to unsigned!!

תבניות ביט מוחדרת
המשמעות משתנה

Sign Extension

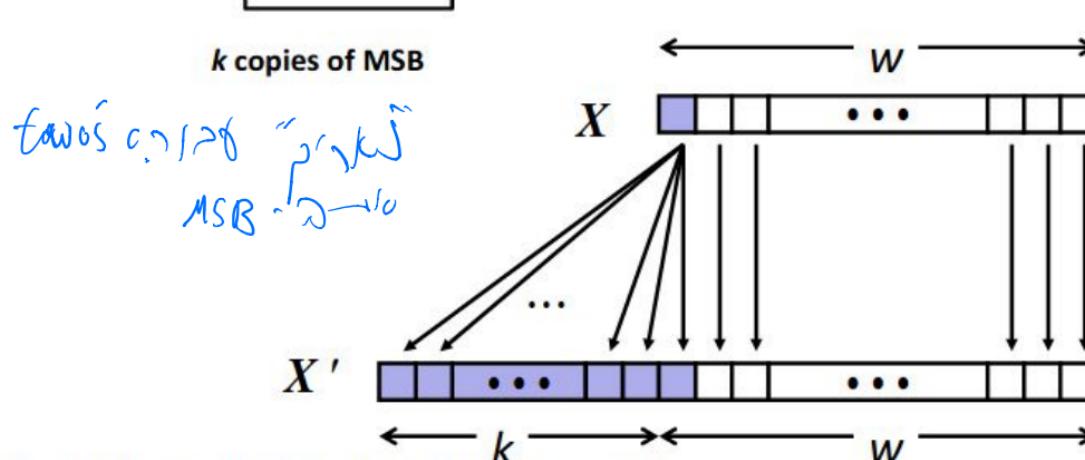
HW'07 Ch 5.1

■ Task:

- Given w -bit signed integer x
- Convert it to $w+k$ -bit integer with same value

■ Rule:

- Make k copies of sign bit:
- $X' = x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0$



Sign Extension: Simple Example

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0
10 =	-32	16	8	4	2

Diagram showing sign extension from a 4-bit value to an 8-bit value. The original 4-bit binary value 1010 is shown in the second row. Red arrows point from the most significant bit (MSB) of this value to the first two bits of the 8-bit result. The 8-bit result has -32 in the 8-bit position and 16 in the 1-bit position.

Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0
-10 =	1	16	8	4	2

Diagram showing sign extension from a 4-bit value to an 8-bit value. The original 4-bit binary value -10 (1010 in two's complement) is shown in the second row. Red arrows point from the most significant bit (MSB) of this value to the first two bits of the 8-bit result. The 8-bit result has 1 in the 8-bit position and 16 in the 1-bit position.

Larger Sign Extension Example

```
short int x = 15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Truncation

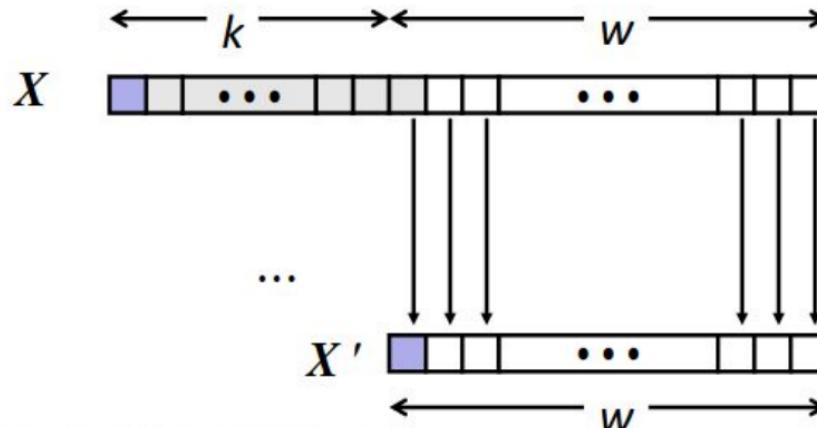
: Up for overflow
for overflow

■ Task:

- Given $k+w$ -bit signed or unsigned integer X
- Convert it to w -bit integer X' with same value for “small enough” X

■ Rule:

- Drop top k bits:
- $X' = x_{w-1}, x_{w-2}, \dots, x_0$



Truncation: Simple Example

(*No sign change*)

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1	
2 =	0	0	1	0	

$$2 \bmod 16 = 2$$

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1	
-6 =	1	0	1	0	

$$-6 \bmod 16 = 26U \bmod 16 = 10U = -6$$

(*Sign change*)

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1	
-6 =	1	0	1	0	

$$10 \bmod 16 = 10U \bmod 16 = 10U = -6$$

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-8	4	2	1	
6 =	0	1	1	0	

$$-10 \bmod 16 = 22U \bmod 16 = 6U = 6$$

Summary:

Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**

- Unsigned: zeros added
- Signed: sign extension
- Both yield expected result

- **Truncating (e.g., unsigned to signed short)**

- Unsigned/signed: bits are truncated
- Result reinterpreted
- Unsigned: mod operation
- Signed: similar to mod
- For small (in magnitude) numbers yields expected behavior

Addition: unsigned

- Standard Addition Function

- Ignores carry output

- Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

unsigned char	1110 1001	E9	223
	+ 1101 0101	+ D5	+ 213
	<hr/>	<hr/>	<hr/>
	1 1011 1110	1BE	446
	<hr/>	<hr/>	<hr/>
	1011 1110	BE	190

Addition: signed

Same operation, different interpretation:

- **TAdd and UAdd have Identical Bit-Level Behavior**

- Signed vs. unsigned addition in C:

```
int s, t, u, v;
```

```
s = (int) ((unsigned) u + (unsigned) v);
```

```
t = u + v
```

- Will give $s == t$

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
<hr/>	<hr/>	<hr/>
1 1011 1110	1BE	446
<hr/>	<hr/>	<hr/>
1011 1110	BE	-66

Power-of-2 Multiply with Shift

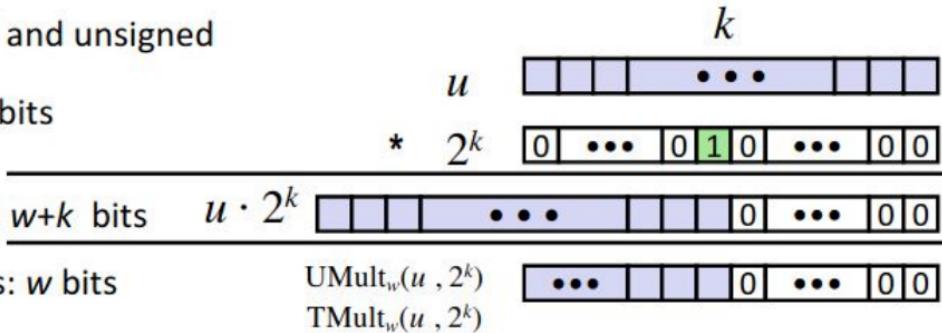
■ Operation

- $u \ll k$ gives $u * 2^k$
- Both signed and unsigned

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



■ Examples

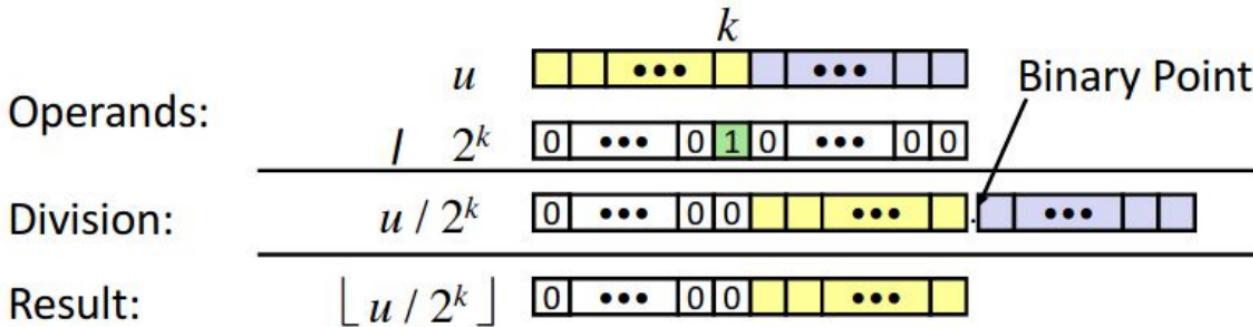
- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- Most machines shift and add faster than multiply
 - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

■ Quotient of Unsigned by Power of 2

- $u \gg k$ gives $\lfloor u / 2^k \rfloor$

- Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Poll: yes/no

1. int a=1; b=(a<<31); c=(b>>31); is a==c?
2. int a=(1<<7)+29; a=(a>>7); a==29?
3. int a=30; a=a>>5; a==0?
4. char a=0300; char b=a>>7; b+1==1?

Complement & Increment Examples

$x = 0$

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~ 0	-1	FF FF	11111111 11111111
$\sim 0 + 1$	0	00 00	00000000 00000000

$x = TMin$

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
$\sim x$	32767	7F FF	01111111 11111111
$\sim x + 1$	-32768	80 00	10000000 00000000

Canonical counter example

Why Should I Use Unsigned? (cont.)

- ***Do Use When Performing Modular Arithmetic***
 - Multiprecision arithmetic
- ***Do Use When Using Bits to Represent Sets***
 - Logical right shift, no sign extension
- ***Do Use In System Programming***
 - Bit masks, device commands,...

Why Should I Use Unsigned?

- *Don't use without understanding implications*

- Easy to make mistakes

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

Bit masking

How do I understand that bit #23, #21 and #2 are on?

```
(a & (1<<23)) && (a & (1<<21)) && (a & (1<<2))
```

How do I set bit #23 and #20?

```
a |= ((1<<23) | (1<<20))
```

Integer C Puzzles

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

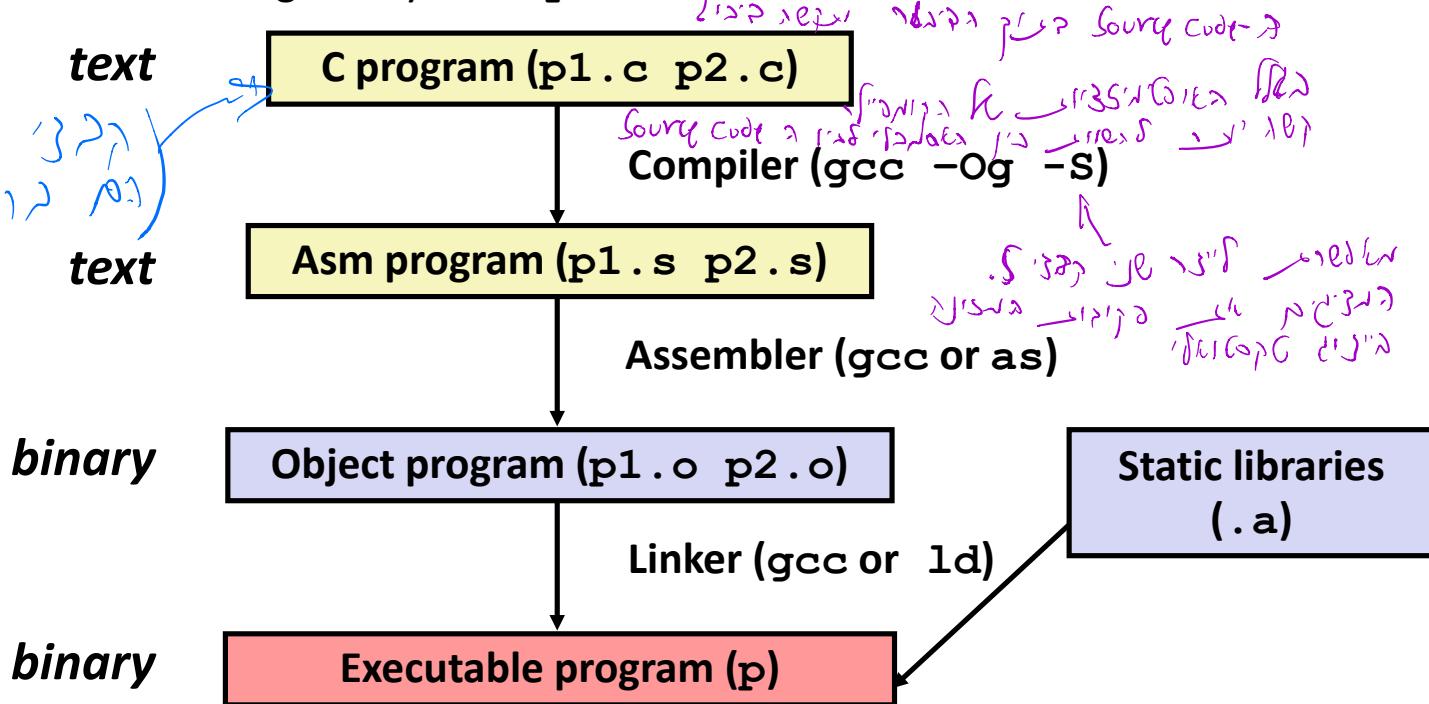
$x < 0$	$\Rightarrow ((x*2) < 0)$	X
$ux \geq 0$		✓
$x \& 7 == 7$	$\Rightarrow (x << 30) < 0$	✓
$ux > -1$		X
$x > y$	$\Rightarrow -x < -y$	X
$x * x \geq 0$		X
$x > 0 \&& y > 0$	$\Rightarrow x + y > 0$	X
$x \geq 0$	$\Rightarrow -x \leq 0$	✓
$x \leq 0$	$\Rightarrow -x \geq 0$	X
$(x -x)>>31 == -1$		X
$ux >> 3 == ux/8$		✓
$x >> 3 == x/8$		X
$x \& (x-1) != 0$		X

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use basic optimizations (`-Og`) [New to recent versions of GCC]
 - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Generated x86-64 Assembly

```
sumstore:  
    pushq   %rbx  
    movq   %rdx, %rbx  
    call    plus  
    movq   %rax, (%rbx)  
    popq   %rbx  
    ret
```

Obtain (on shark machine) with command

(!>5 _m4e r4 p4 m4 n4 s4 t4 s4 u4 v4 w4)

```
gcc -Og -S sum.c
```

Produces file **sum.s**

Warning: Will get very different results on non-Shark machines (Andrew Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

Sum 5: What it really looks like

```
.globl sumstore
.type sumstore, @function

sumstore:
.LFB35:
    .cfi_startproc
    pushq %rbx
    .cfi_offset 16
    .cfi_offset 3, -16
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    .cfi_offset 8
    ret
    .cfi_endproc

.LFE35:
.size sumstore, .-sumstore
```

(first three lines are valid code)
following CFI directives help
with register management

What it really looks like

```
.globl sumstore
.type sumstore, @function
sumstore:
.LFB35:
.cfi_startproc
    pushq %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    .cfi_def_cfa_offset 8
    ret
.cfi_endproc
.LFE35:
.size sumstore, .-sumstore
```

Things that look weird
and are preceded by a '
are generally directives.

sumstore:

```
pushq %rbx
movq %rdx, %rbx
call plus
movq %rax, (%rbx)
popq %rbx
ret
```

Assembly Characteristics: Data Types

■ “Integer” data of 1, 2, 4, or 8 bytes

- Data values
- Addresses (untyped pointers)

(*int* ↳ *DataTypes*.*is*)
is Char, int, ...)

■ Floating point data of 4, 8, or 10 bytes

} *f32 f64*

■ (SIMD vector data types of 8, 16, 32 or 64 bytes)

: *v16f32 v16f64*

■ Code: Byte sequences encoding series of instructions

■ No aggregate types such as arrays or structures

- Just contiguously allocated bytes in memory

(*Storage* ↳ *int* ↳ *bytes* ↳ *bytē - byte* ↳ *ROM*)

Assembly Characteristics: Operations

Transfer data between memory and register
Perform arithmetic function on register or memory data
Transfer control

■ Transfer data between memory and register

- Load data from memory into register (Load Address)
- Store register data into memory (Store Address)

■ Perform arithmetic function on register or memory data

■ Transfer control

- Unconditional jumps to/from procedures
- Conditional branches

Object Code

: binary \rightarrow $\text{S} \rightarrow$ P

Code for sumstore

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

■ Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

■ Linker

- Resolves references between files
- Combines with static run-time libraries
 - E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

t *dest*

dest *W38 3919*

0x40059e:	48	89	03
-----------	----	----	----

■ C Code

- Store value **t** where designated by **dest**

■ Assembly

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:
 - t:** Register **%rax**
 - dest:** Register **%rbx**
 - *dest:** Memory **M[%rbx]**

■ Object Code - *binary*

- 3-byte instruction
- Stored at address **0x40059e**

Disassembling Object Code

('fənɛm-đ bɪnərɪ ɔ:səm)
(Ləndz'f ɔ:səm ɔ:nə)

Disassembled

```
0000000000400595 <sumstore>:  
400595: 53                      push    %rbx  
400596: 48 89 d3                mov     %rdx,%rbx  
400599: e8 f2 ff ff ff        callq   400590 <plus>  
40059e: 48 89 03                mov     %rax,(%rbx)  
4005a1: 5b                      pop    %rbx  
4005a2: c3                      retq
```

- **Disassembler** (məsəm'bərl)
objdump -d sum (ɔ:b'dʌmp dʒʌm)

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a .out (complete executable) or .o file

Alternate Disassembly

Disassembled

```
Dump of assembler code for function sumstore:  
0x0000000000400595 <+0>: push    %rbx  
0x0000000000400596 <+1>: mov     %rdx,%rbx  
0x0000000000400599 <+4>: callq   0x400590 <plus>  
0x000000000040059e <+9>: mov     %rax,(%rbx)  
0x00000000004005a1 <+12>:pop    %rbx  
0x00000000004005a2 <+13>:retq
```

GDB → rd fjs pjs hjs

■ Within gdb Debugger

- Disassemble procedure

gdb sum

disassemble sumstore

Alternate Disassembly

Object Code

```
0x0400595:  
0x53  
0x48  
0x89  
0xd3  
0xe8  
0xf2  
0xff  
0xff  
0xff  
0x48  
0x89  
0x03  
0x5b  
0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:  
0x0000000000400595 <+0>: push    %rbx  
0x0000000000400596 <+1>: mov     %rdx,%rbx  
0x0000000000400599 <+4>: callq   0x400590 <plus>  
0x000000000040059e <+9>: mov     %rax,(%rbx)  
0x00000000004005a1 <+12>:pop    %rbx  
0x00000000004005a2 <+13>:retq
```

■ Within gdb Debugger

- Disassemble procedure
`gdb sum`
`disassemble sumstore`
- Examine the 14 bytes starting at `sumstore`
`x/14xb sumstore`

What Can be Disassembled? *gcc -Og,*

```
% objdump -d WINWORD.EXE  
  
WINWORD.EXE:      file format pei-i386  
  
No symbols in "WINWORD.EXE".  
Disassembly of section .text:  
  
30001000 <.text>:  
30001000:  
30001001:  
30001003:  
30001005:  
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

(.RCS P3272 Source code always 10f)

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- C, assembly, machine code

Levels of Abstraction

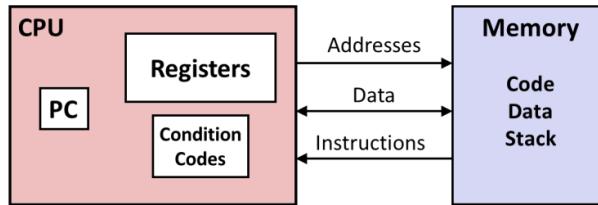
C programmer

C code

layers

Assembly programmer

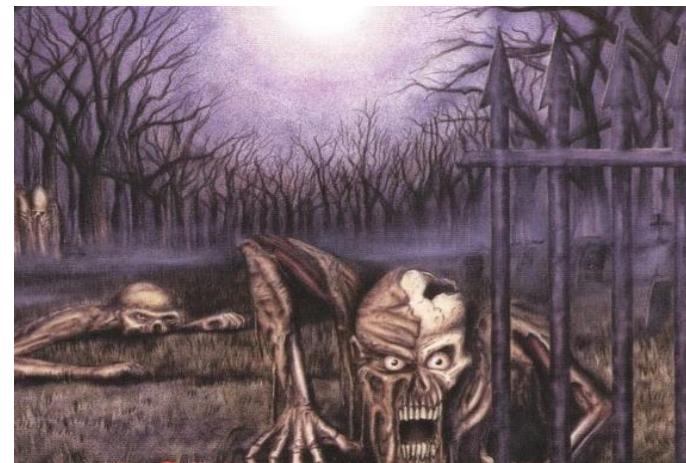
layers



Computer Designer

(Micro - Eng)

Caches, clock freq, layout, ...



Of course, you know that: It's why you are taking this course.

Definitions

(جُنْدِلِ رَوْنِيْزِيْنْ)

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing correct machine/assembly code

- Examples: instruction set specification, registers
- **Machine Code:** The byte-level programs that a processor executes
- **Assembly Code:** A text representation of machine code

- **Microarchitecture:** Implementation of the architecture

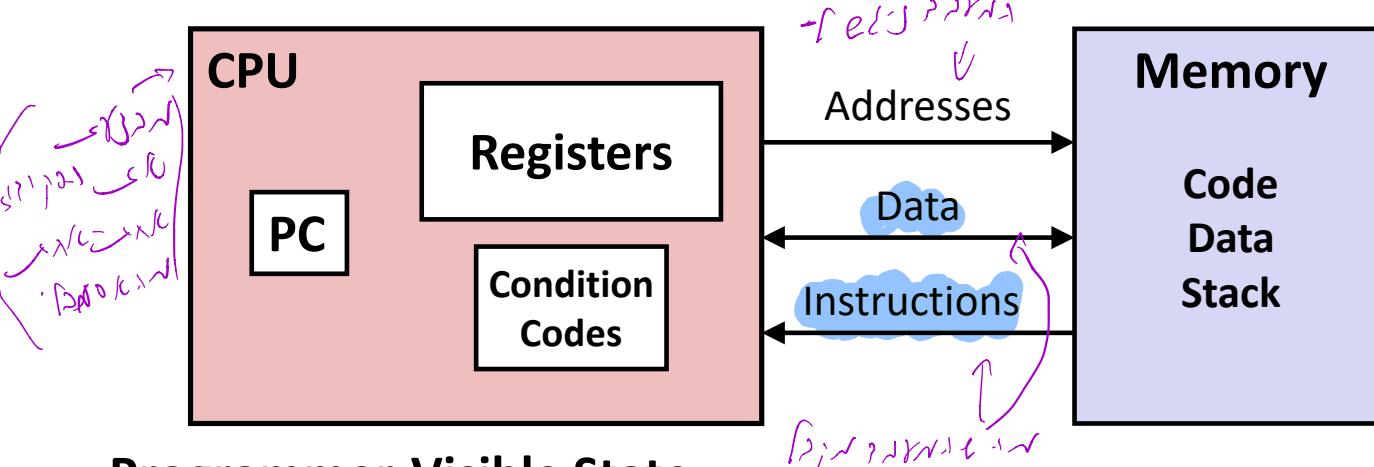
- Examples: cache sizes and core frequency

(أَنْدِرُوْنِيْزِيْنْ تَحْتِيْنِيْزِيْنْ)

- **Example ISAs:**

- Intel: x86, IA32, Itanium, x86-64
- ARM: Used in almost all mobile phones
- RISC V: New open-source ISA

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called “RIP” (x86-64)
- **Register file**
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

Assembly Characteristics: Data Types

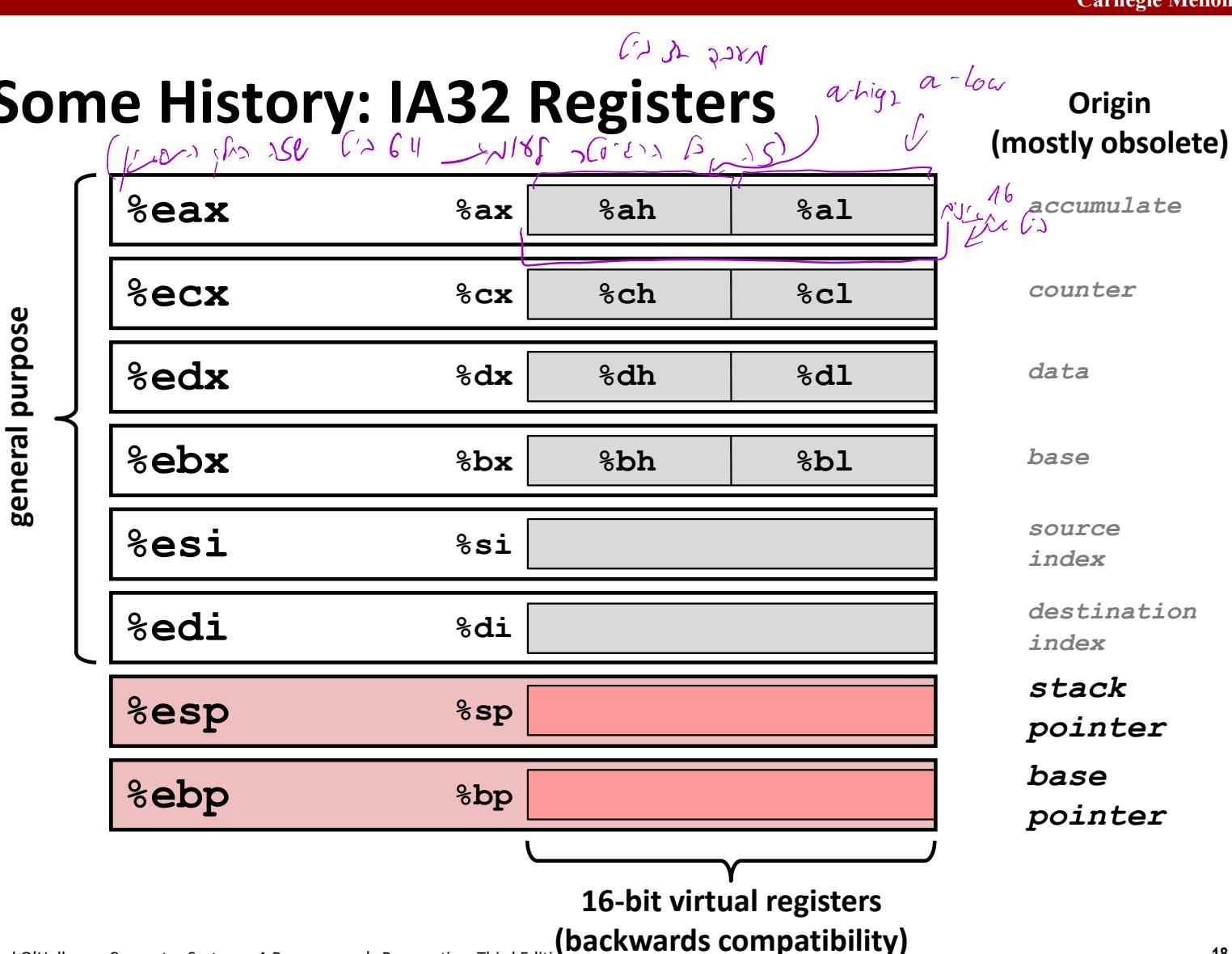
- “Integer” data of 1, 2, 4, or 8 bytes
 - Data values
 - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- (SIMD vector data types of 8, 16, 32 or 64 bytes)
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
 - Just contiguously allocated bytes in memory

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers



Assembly Characteristics: Operations

- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
 - Unconditional jumps to/from procedures
 - Conditional branches
 - Indirect branches

Moving Data

Moving Data

~~movq Source, Dest~~

~~~~~

~~Load - Value~~  
~~Push Value~~

~~AT&T notation~~

~~?Source, dest~~

## Operand Types

### Immediate:

- Constant integer data
- Example: \$0x400, \$-533
- Like C constant, but prefixed with '\$'
- Encoded with 1, 2, or 4 bytes

### Register:

- One of 16 integer registers
- Example: %rax, %r13

- But %rsp reserved for special use

- Others have special uses for particular instructions

### Memory

- 8 consecutive bytes of memory at address given by register
- Simplest example: (%rax)

- Various other “addressing modes”

%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

%rbp

%rN

**Warning: Intel docs use  
mov Dest, Source**

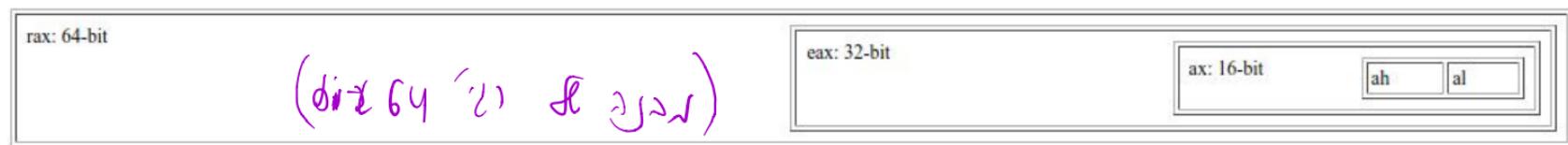
# movq Operand Combinations

|                                                                  | Source | Dest | Src,Dest                                            | C Analog       |
|------------------------------------------------------------------|--------|------|-----------------------------------------------------|----------------|
| movq                                                             | Imm    | Reg  | <i>(Pion piissi fl "ejenay?)</i><br>movq \$0x4,%rax | temp = 0x4;    |
|                                                                  |        | Mem  | movq \$-147,(%rax)                                  | *p = -147;     |
|                                                                  | Reg    | Reg  | movq %rax,%rdx                                      | temp2 = temp1; |
|                                                                  | Mem    | Reg  | movq %rax,(%rdx)                                    | *p = temp;     |
|                                                                  | Mem    | Reg  | movq (%rax),%rdx                                    | temp = *p;     |
| <del>④ movq (%rax),(%rdx)</del><br><i>if 2nd operand is zero</i> |        |      |                                                     |                |

**Cannot do memory-memory transfer with a single instruction**

# Poll Yes/No

- Registers are part of memory *n ✓*
- Every register can be used only for specific purpose *n ✓*
- Is the following picture correctly represents RAX/EAX/AX/AH/AL ? *y ✓*



- If `rax` holds a memory address, then the results of this operation depends on the pointer type *n ✓*

`add $1, %rax`

# Simple Memory Addressing Modes

I

## Normal (R)

- Register R specifies memory address
- Aha! Pointer dereferencing in C

`movq (%rcx), %rax`

**Mem[Reg[R]]**

(rcx → register address → base address)

→ mem[Reg[rcx]] ↪

II

## Displacement D(R)

**Mem[Reg[R]+D]**

- Register R specifies start of memory region
- Constant displacement D specifies offset

`movq 8(%rbp), %rdx`

(rbp → base register → base address)

# Example of Simple Addressing Modes

```
void whatAmI(<type> a, <type> b)
{
    ???
}
```

$\text{tmp1} = *a$   
 $\text{tmp2} = *b$   
 $*a = \text{tmp2}$   
 $*b = \text{tmp1}$

%rdi

**whatAmI: Swap**      a      tmp1  
 ||  
 movq (%rdi), %rax  
 movq (%rsi), %rdx = tmp2  
 movq %rdx, (%rdi)  
 movq %rax, (%rsi)  
 ret

# Example of Simple Addressing Modes

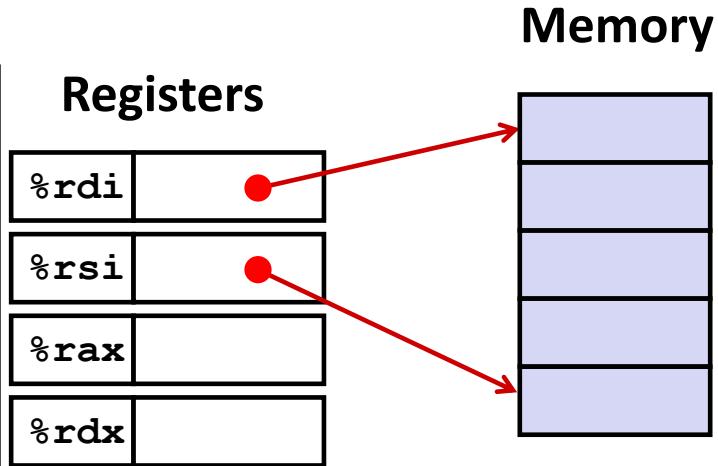
```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

|      |              |
|------|--------------|
| movq | (%rdi), %rax |
| movq | (%rsi), %rdx |
| movq | %rdx, (%rdi) |
| movq | %rax, (%rsi) |
| ret  |              |

# Understanding Swap()

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



| Register | Value |
|----------|-------|
| %rdi     | xp    |
| %rsi     | yp    |
| %rax     | t0    |
| %rdx     | t1    |

swap:

```
    movq    (%rdi), %rax # t0 = *xp
    movq    (%rsi), %rdx # t1 = *yp
    movq    %rdx, (%rdi) # *xp = t1
    movq    %rax, (%rsi) # *yp = t0
    ret
```

(→)

(↑)

(→)

# Understanding Swap()

Registers

|      |       |
|------|-------|
| %rdi | 0x120 |
| %rsi | 0x100 |
| %rax |       |
| %rdx |       |

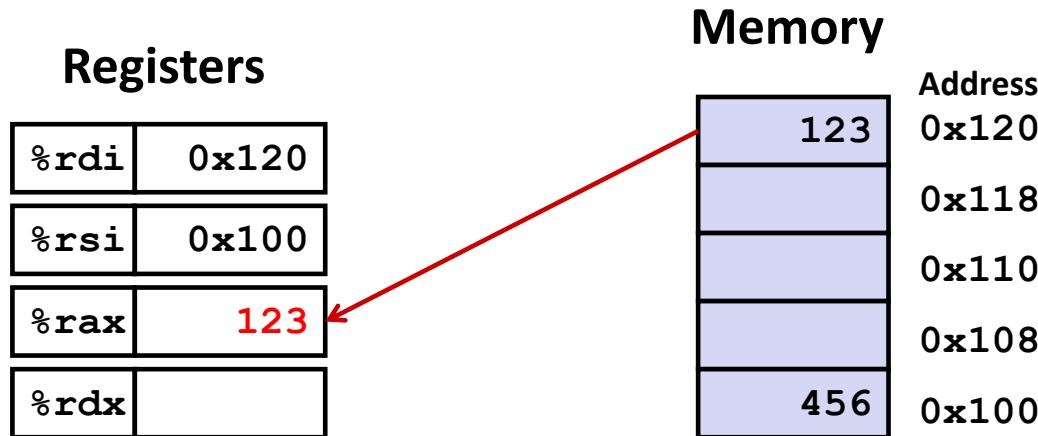
Memory

|     |                  |
|-----|------------------|
| 123 | Address<br>0x120 |
|     | 0x118            |
|     | 0x110            |
|     | 0x108            |
| 456 | 0x100            |

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

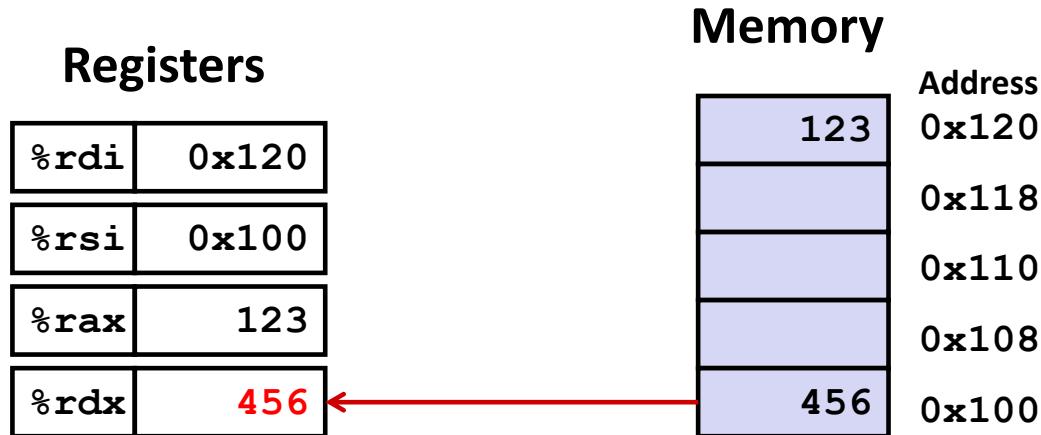
# Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

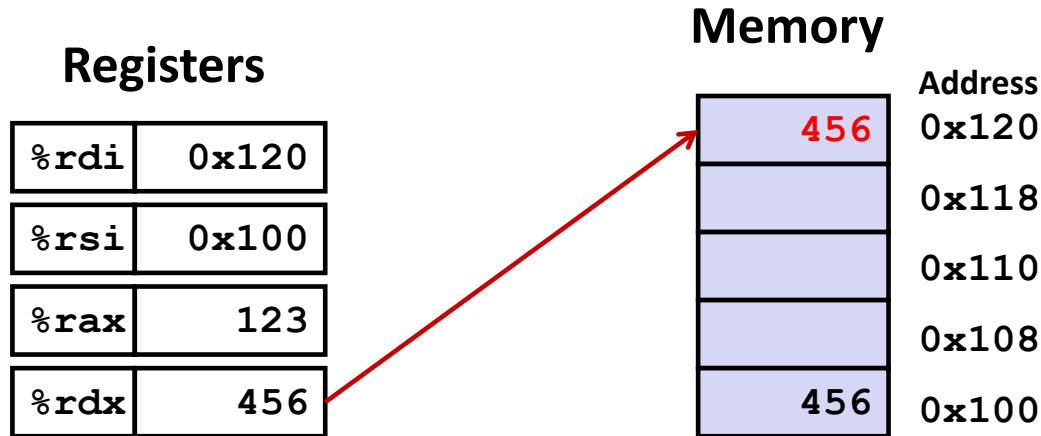
# Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

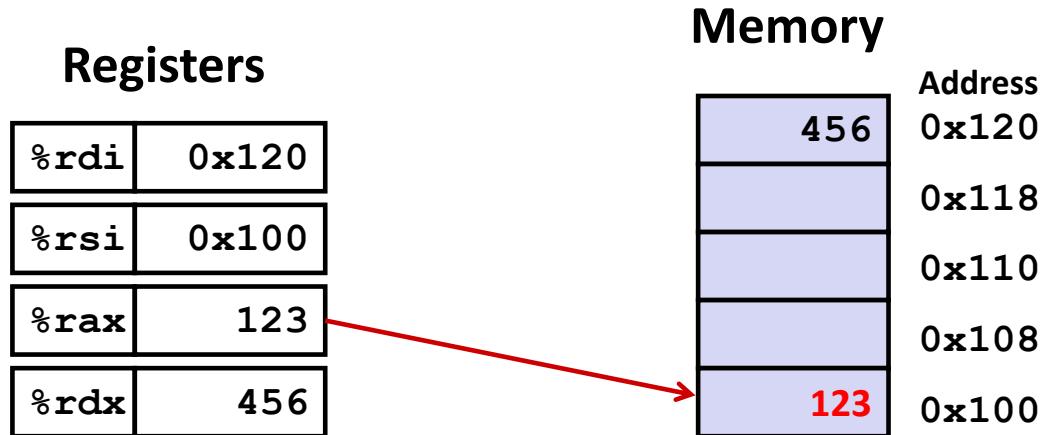
# Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

# Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

# Simple Memory Addressing Modes

## ■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx), %rax
```

## ■ Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp), %rdx
```

# Complete Memory Addressing Modes

## ■ Most General Form

$D(Rb, Ri, S)$

$\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri] + D]$

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

(*العنوان المطلق*) *هي* *العنوان* *لـ* *العنوان* *المطلق* *العنوان* *المطلق*

## ■ Special Cases

$(Rb, Ri)$

$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]$

$D(Rb, Ri)$

$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]$

$(Rb, Ri, S)$

$\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri]]$

# Address Computation Examples

|                   |        |
|-------------------|--------|
| <code>%rdx</code> | 0xf000 |
| <code>%rcx</code> | 0x0100 |

 $D(Rb, Ri, S)$  $\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri] + D]$ 

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

| Expression                   | Address Computation   | Address |
|------------------------------|-----------------------|---------|
| <code>0x8(%rdx)</code>       | $0xf000 + 0x8$        | 0xf008  |
| <code>(%rdx, %rcx)</code>    | $0xf000 + 0x0100$     | 0xf100  |
| <code>(%rdx, %rcx, 4)</code> | $0xf000 + 4 * 0x0100$ | 0xf400  |
| <code>0x80(,%rdx,2)</code>   |                       | 0x7f080 |

# Address Computation Examples

|      |        |
|------|--------|
| %rdx | 0xf000 |
| %rcx | 0x0100 |

| Expression    | Address Computation | Address |
|---------------|---------------------|---------|
| 0x8(%rdx)     | 0xf000 + 0x8        | 0xf008  |
| (%rdx,%rcx)   | 0xf000 + 0x100      | 0xf100  |
| (%rdx,%rcx,4) | 0xf000 + 4*0x100    | 0xf400  |
| 0x80(,%rdx,2) | 2*0xf000 + 0x80     | 0x1e080 |

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

# Address Computation Instruction

*(Reg<sub>1</sub> + S · Reg<sub>2</sub> + D)*

## ■ leaq Src, Dst

- Src is address mode expression
- Set Dst to address denoted by expression

## ■ Uses

- Computing addresses without a memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form  $x + k^*y$ 
  - $k = 1, 2, 4, \text{ or } 8$

## ■ Example

```
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t = x+2*x
salq $2, %rax           # return t<<2
```

*↳ 3 -> 120001*

# Some Arithmetic Operations

## ■ Two Operand Instructions:

| <i>Format</i>         | <i>Computation</i> |                         |
|-----------------------|--------------------|-------------------------|
| addq <i>Src,Dest</i>  | Dest = Dest + Src  |                         |
| subq <i>Src,Dest</i>  | Dest = Dest – Src  |                         |
| imulq <i>Src,Dest</i> | Dest = Dest * Src  |                         |
| salq <i>Src,Dest</i>  | Dest = Dest << Src | <i>Also called shlq</i> |
| sarq <i>Src,Dest</i>  | Dest = Dest >> Src | <i>Arithmetic</i>       |
| shrq <i>Src,Dest</i>  | Dest = Dest >> Src | <i>Logical</i>          |
| xorq <i>Src,Dest</i>  | Dest = Dest ^ Src  |                         |
| andq <i>Src,Dest</i>  | Dest = Dest & Src  |                         |
| orq <i>Src,Dest</i>   | Dest = Dest   Src  |                         |

- Watch out for argument order! *Src,Dest*  
(Warning: Intel docs use “op *Dest,Src*”)
- No distinction between signed and unsigned int (why?)

# Poll Yes/No

1. leaq 0x1000(%rax, \$0x20, 2), %rbx
- If rbx is 0x1040, then  $\text{rax} == 0$
2. addq \$2, 0x1000(%rax) is invalid
- Src*      *dest*
- $0x1000 - (%rax) = 0x40 = (%rbx)$
- $\Rightarrow$

# Some Arithmetic Operations

## ■ One Operand Instructions

|      |             |                    |
|------|-------------|--------------------|
| incq | <i>Dest</i> | $Dest = Dest + 1$  |
| decq | <i>Dest</i> | $Dest = Dest - 1$  |
| negq | <i>Dest</i> | $Dest = -Dest$     |
| notq | <i>Dest</i> | $Dest = \sim Dest$ |

## ■ See book for more instructions

# Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:

```
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq   %rcx, %rax
    ret
```

## Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once

# Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:

```
leaq    (%rdi,%rsi), %rax    # t1
addq    %rdx, %rax          # t2
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx            # t4
leaq    4(%rdi,%rdx), %rcx  # t5
imulq   %rcx, %rax          # rval
ret
```

| Register | Use(s)                           |
|----------|----------------------------------|
| %rdi     | Argument <b>x</b>                |
| %rsi     | Argument <b>y</b>                |
| %rdx     | Argument <b>z</b> ,<br><b>t4</b> |
| %rax     | <b>t1, t2, rval</b>              |
| %rcx     | <b>t5</b>                        |

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

# Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

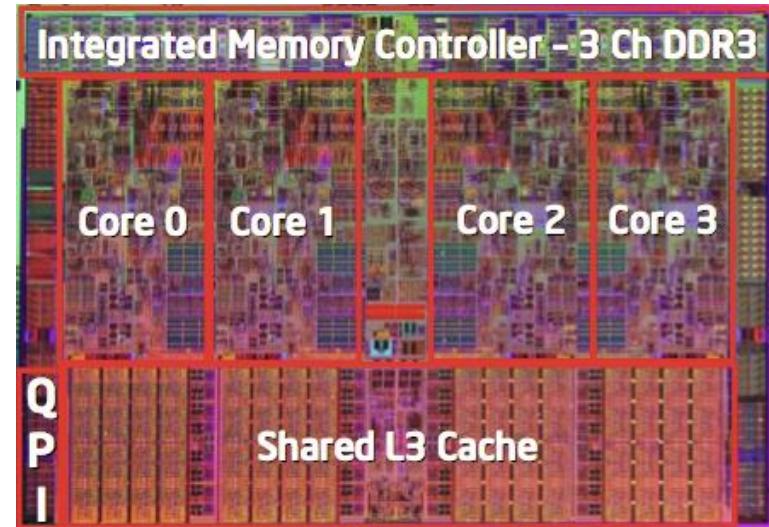
# Intel x86 Evolution: Milestones

| <i>Name</i>         | <i>Date</i> | <i>Transistors</i> | <i>MHz</i>                                                                                                                                                      |
|---------------------|-------------|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ■ <b>8086</b>       | <b>1978</b> | <b>29K</b>         | <b>5-10</b>                                                                                                                                                     |
|                     |             |                    | <ul style="list-style-type: none"><li>▪ First 16-bit Intel processor. Basis for IBM PC &amp; DOS</li><li>▪ 1MB address space</li></ul>                          |
| ■ <b>386</b>        | <b>1985</b> | <b>275K</b>        | <b>16-33</b>                                                                                                                                                    |
|                     |             |                    | <ul style="list-style-type: none"><li>▪ First 32 bit Intel processor , referred to as IA32</li><li>▪ Added “flat addressing”, capable of running Unix</li></ul> |
| ■ <b>Pentium 4E</b> | <b>2004</b> | <b>125M</b>        | <b>2800-3800</b>                                                                                                                                                |
|                     |             |                    | <ul style="list-style-type: none"><li>▪ First 64-bit Intel x86 processor, referred to as x86-64</li></ul>                                                       |
| ■ <b>Core 2</b>     | <b>2006</b> | <b>291M</b>        | <b>1060-3333</b>                                                                                                                                                |
|                     |             |                    | <ul style="list-style-type: none"><li>▪ First multi-core Intel processor</li></ul>                                                                              |
| ■ <b>Core i7</b>    | <b>2008</b> | <b>731M</b>        | <b>1600-4400</b>                                                                                                                                                |
|                     |             |                    | <ul style="list-style-type: none"><li>▪ Four cores (our shark machines)</li></ul>                                                                               |

# Intel x86 Processors, cont.

## ■ Machine Evolution

|                   |      |      |
|-------------------|------|------|
| ■ 386             | 1985 | 0.3M |
| ■ Pentium         | 1993 | 3.1M |
| ■ Pentium/MMX     | 1997 | 4.5M |
| ■ PentiumPro      | 1995 | 6.5M |
| ■ Pentium III     | 1999 | 8.2M |
| ■ Pentium 4       | 2000 | 42M  |
| ■ Core 2 Duo      | 2006 | 291M |
| ■ Core i7         | 2008 | 731M |
| ■ Core i7 Skylake | 2015 | 1.9B |



## ■ Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

# Intel x86 Processors, cont.

## ■ Past Generations

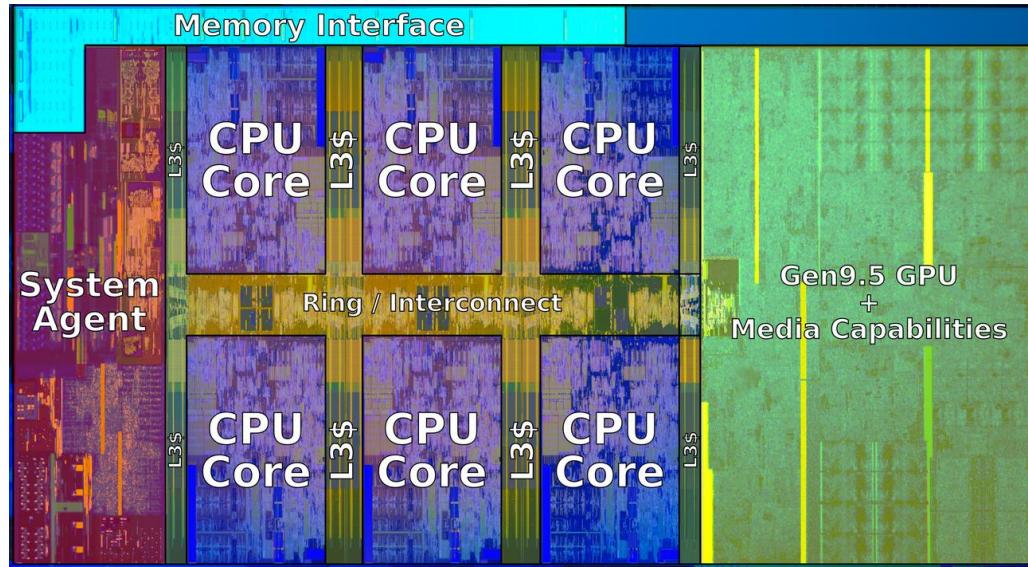
|   |                             | Process technology |
|---|-----------------------------|--------------------|
| ■ | 1 <sup>st</sup> Pentium Pro | 1995               |
| ■ | 1 <sup>st</sup> Pentium III | 1999               |
| ■ | 1 <sup>st</sup> Pentium 4   | 2000               |
| ■ | 1 <sup>st</sup> Core 2 Duo  | 2006               |
|   |                             | 600 nm             |
|   |                             | 250 nm             |
|   |                             | 180 nm             |
|   |                             | 65 nm              |

## ■ Recent & Upcoming Generations

|    |              |       |       |
|----|--------------|-------|-------|
| 1. | Nehalem      | 2008  | 45 nm |
| 2. | Sandy Bridge | 2011  | 32 nm |
| 3. | Ivy Bridge   | 2012  | 22 nm |
| 4. | Haswell      | 2013  | 22 nm |
| 5. | Broadwell    | 2014  | 14 nm |
| 6. | Skylake      | 2015  | 14 nm |
| 7. | Kaby Lake    | 2016  | 14 nm |
| 8. | Coffee Lake  | 2017  | 14 nm |
| ■  | Cannon Lake  | 2019? | 10 nm |

**Process technology dimension**  
= width of narrowest wires  
(10 nm ≈ 100 atoms wide)

# 2018 State of the Art: Coffee Lake



## ■ Mobile Model: Core i7

- 2.2-3.2 GHz
- 45 W

## ■ Desktop Model: Core i7

- Integrated graphics
- 2.4-4.0 GHz
- 35-95 W

## ■ Server Model: Xeon E

- Integrated graphics
- Multi-socket enabled
- 3.3-3.8 GHz
- 80-95 W

# x86 Clones: Advanced Micro Devices (AMD)

## ■ Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

## ■ Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

## ■ Recent Years

- Intel got its act together
  - Leads the world in semiconductor technology
- AMD has fallen behind
  - Relies on external semiconductor manufacturer

# Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode

# Our Coverage

## ■ IA32

- The traditional x86
- For 15/18-213: RIP, Summer 2015

## ■ x86-64

- The standard
- shark> gcc hello.c
- shark> gcc -m64 hello.c

## ■ Presentation

- Book covers x86-64
- Web aside on IA32
- We will only cover x86-64

D

KUKU.H

```
int foo (int x);
```

C

KUKU.C

```
#include "KUKU.h"
```

```
int foo (int x)
```

{

S

E

A.C

~~KUKU~~

#include

"KUKU.h"

}

foo(s);

3

• b 121xJ include 7or xJ → 7e7s Jypt" C 377  
• i 77'77 7eww 7sw .h 777.

# Today

## ■ Procedures

- Mechanisms
- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

Basic Note on Procedure Based on Self-study 6 p  
(P. 2103)

# Mechanisms in Procedures

## ■ Passing control

- To beginning of procedure code
- Back to return point

## ■ Passing data

- Procedure arguments
- Return value

## ■ Memory management

- Allocate during procedure execution
- Deallocate upon return

## ■ Mechanisms all implemented with machine instructions

## ■ x86-64 implementation of a procedure uses only those mechanisms required

```
P (...) {
    •
    •
    y = Q(x);
    print(y)
    •
}
```

```
int Q(int i)
{
    int t = 3*i;
    int v[10];
    •
    return v[t];
}
```

# Mechanisms in Procedures

## ■ Passing control

- To beginning of procedure code
- Back to return point

## ■ Passing data

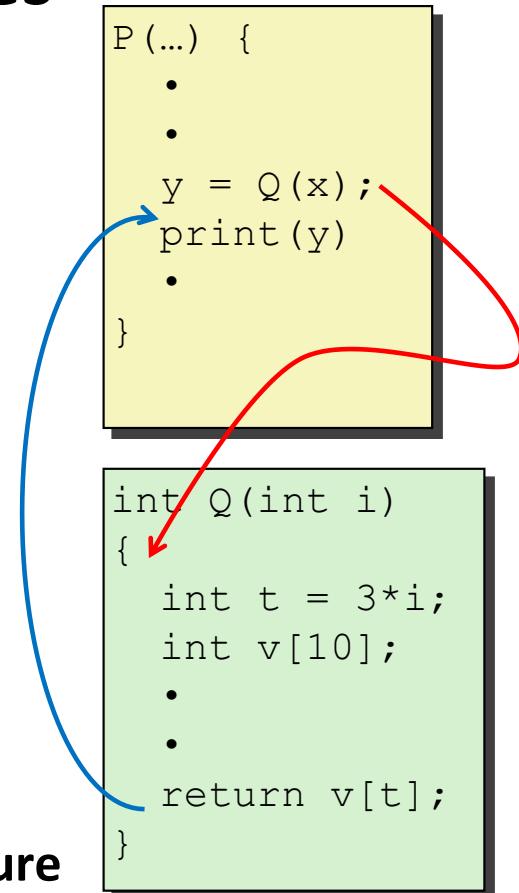
- Procedure arguments
- Return value

## ■ Memory management

- Allocate during procedure execution
- Deallocate upon return

## ■ Mechanisms all implemented with machine instructions

## ■ x86-64 implementation of a procedure uses only those mechanisms required



# Mechanisms in Procedures

## ■ Passing control

- To beginning of procedure code
- Back to return point

## ■ Passing data

- Procedure arguments
- Return value

## ■ Memory management

- Allocate during procedure execution
- Deallocate upon return

## ■ Mechanisms all implemented with machine instructions

## ■ x86-64 implementation of a procedure uses only those mechanisms required

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

# Mechanisms in Procedures

## ■ Passing control

- To beginning of procedure code
- Back to return point

## ■ Passing data

- Procedure arguments
- Return value

## ■ Memory management

- Allocate during procedure execution
- Deallocate upon return

## ■ Mechanisms all implemented with machine instructions

## ■ x86-64 implementation of a procedure uses only those mechanisms required

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

# Mechanisms in Procedures

```
P(...){
```

Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the **Application Binary Interface**

(ABI).  
(App Binary Interface)

- Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
int v[10];
•
•
return v[t];
}
```

# Today

## ■ Procedures

- Mechanisms
- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# x86-64 Stack

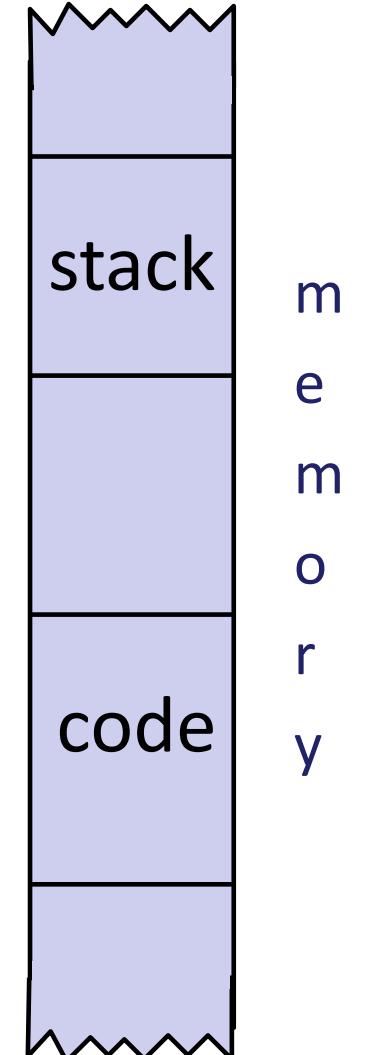
## ■ Region of memory managed with stack discipline

- Memory viewed as array of bytes.
- Different regions have different purposes.
- (Like ABI, a policy decision)

(overlaid on the stack →)

process stack

process stack



# x86-64 Stack

- Region of memory managed with stack discipline

FIFO

(first in  
last out)

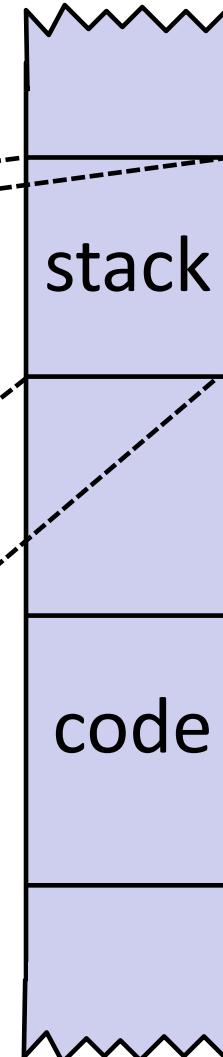
(last in  
first out)

Stack Pointer: %rsp →

Stack “Bottom”

Stack “Top”

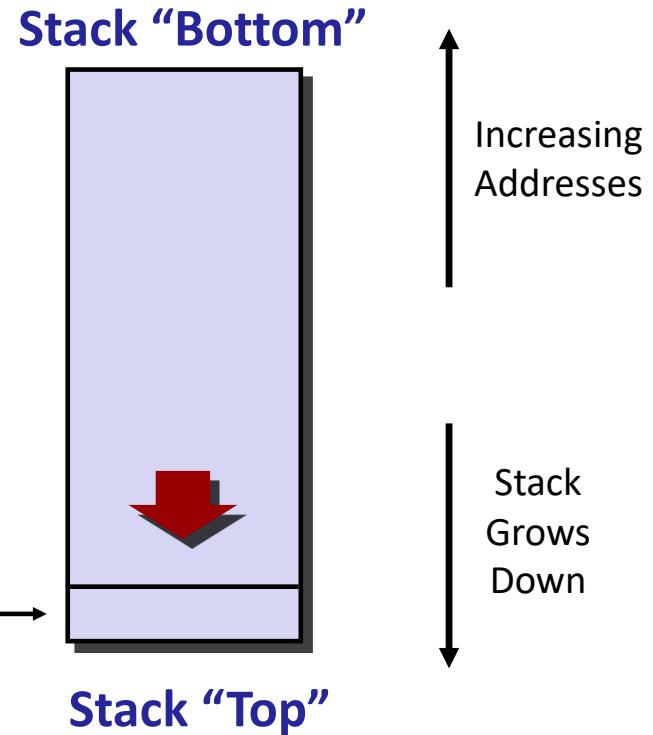
(last in  
last out)



# x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of “top” element

Stack Pointer: `%rsp` →



# x86-64 Stack: Push

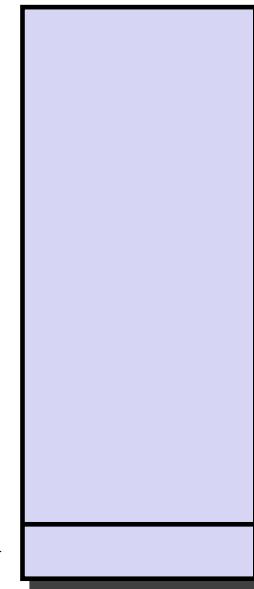
## ■ **pushq Src**

- Fetch operand at *Src*
- Decrement **%rsp** by 8
- Write operand at address given by **%rsp**

( *Stack grows down* )  
**Stack Pointer: %rsp** →

val

Stack “Bottom”



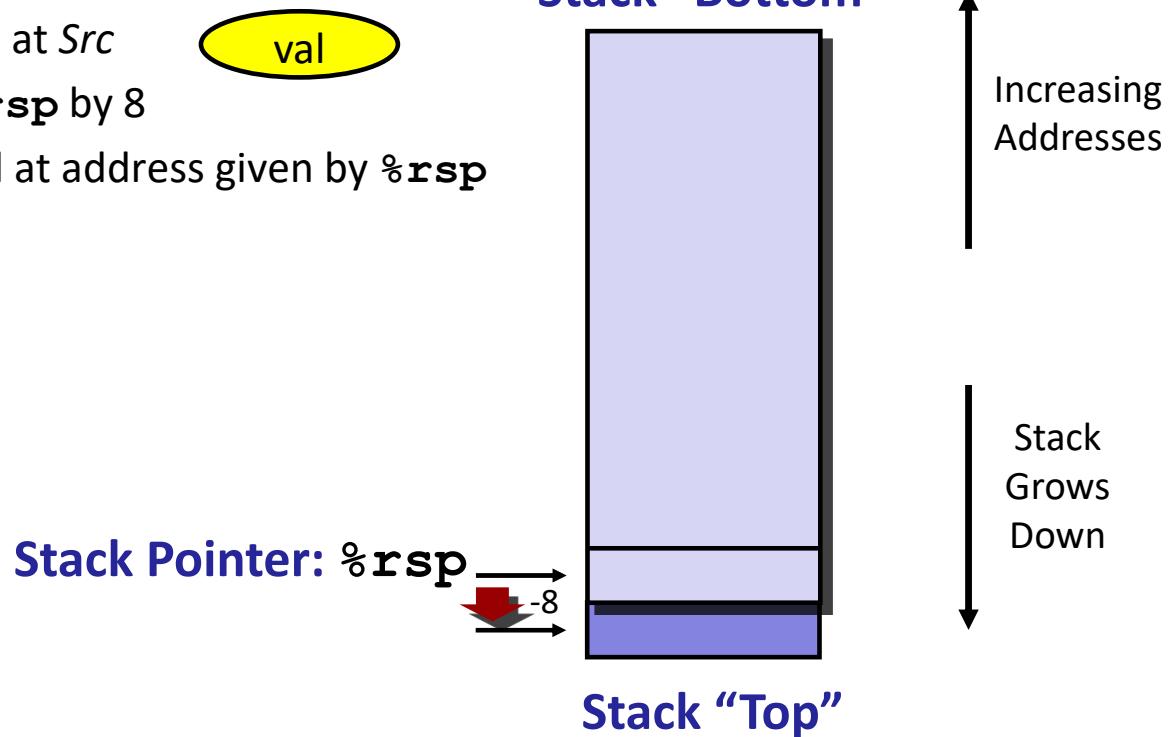
Increasing  
Addresses

Stack  
Grows  
Down

# x86-64 Stack: Push

## ■ `pushq Src`

- Fetch operand at *Src*
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



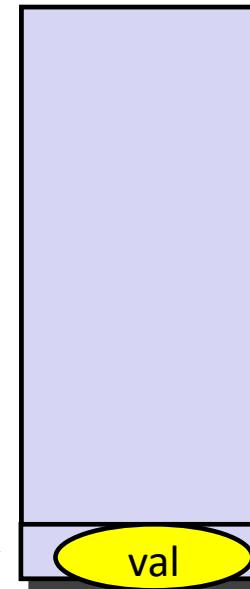
# x86-64 Stack: Pop

## ■ **popq Dest**

- Read value at address given by `%rsp`
- Increment `%rsp` by 8 *(word in X86-64)*
- Store value at Dest (usually a register)

Stack Pointer: `%rsp` →

Stack “Bottom”



Stack “Top”

# x86-64 Stack: Pop

## ■ **popq Dest**

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (usually a register)



**Stack Pointer: %rsp**  $\xrightarrow{+8}$

**Stack “Bottom”**



**Stack “Top”**

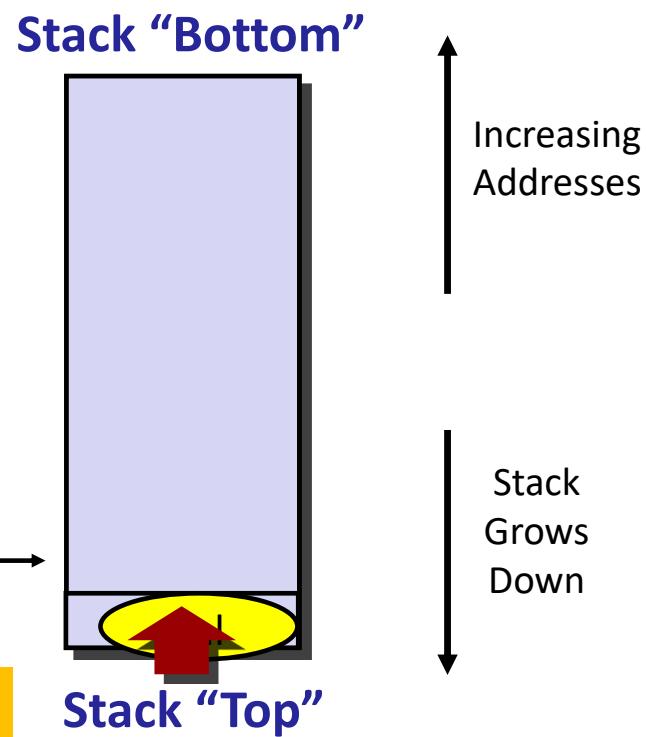
# x86-64 Stack: Pop

■ popq *Dest*

- Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

## Stack Pointer: %rsp →

(The memory doesn't change,  
only the value of %rsp)



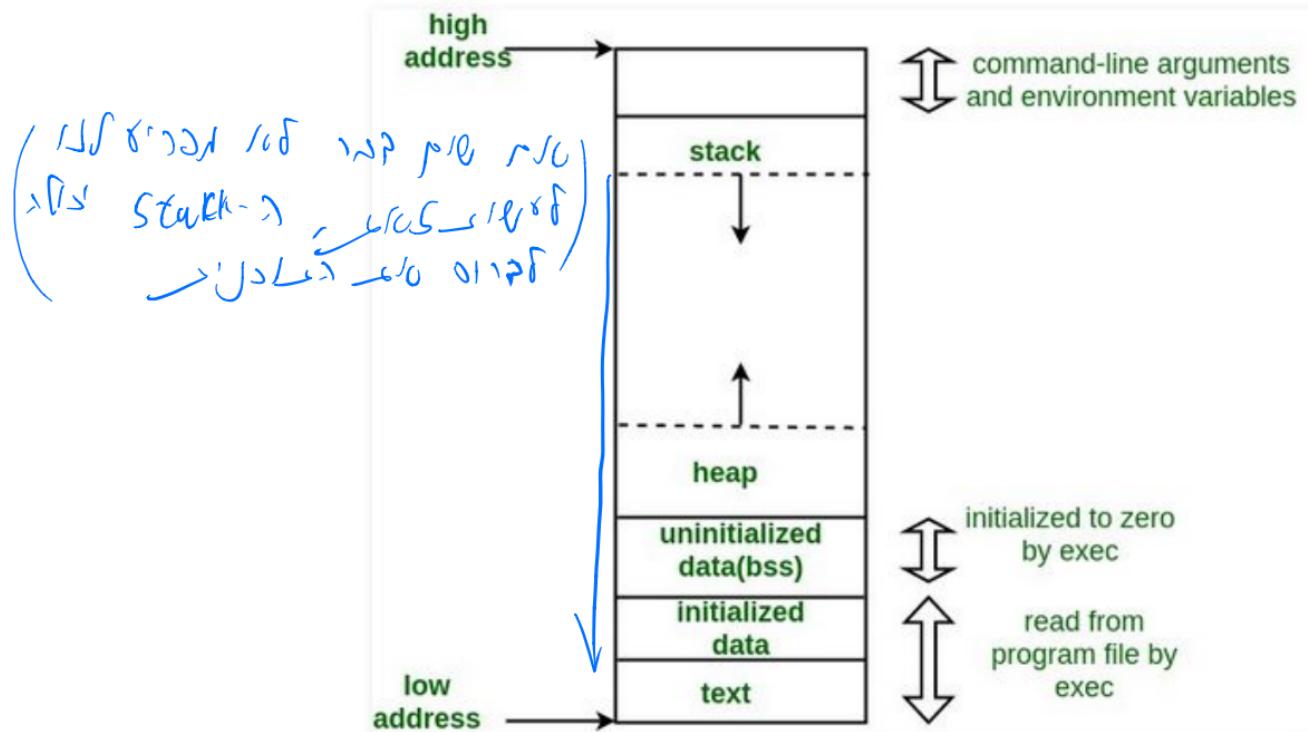
# Poll yes/no

1. Stack is a region of memory used for temporary variables and function calls ✓
2. One cannot use `movq` to access stack memory ✓
3. `%rsp` holds the pointer to the most recent valid value on the stack ✓
4. `popq` increments `%rsp`, `pushq` decrements `%rsp` ✓
5. Both `popq` and `pushq` change the contents of stack memory ✓
6. Pushing too much data on stack may overwrite the program instructions ✓

NO

(Yes, `pushq` does not write to memory, it just pushes the value)

# Typical program memory layout



# Today

## ■ Procedures

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

# Code Examples

```
void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
    400540: push    %rbx          # Save %rbx
    400541: mov     %rdx,%rbx    # Save dest
    400544: callq   400550 <mult2> # mult2(x,y)
    400549: mov     %rax,(%rbx)  # Save at dest
    40054c: pop    %rbx          # Restore %rbx
    40054d: retq               # Return
```

```
long mult2(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
    400550: mov     %rdi,%rax    # a
    400553: imul   %rsi,%rax    # a * b
    400557: retq               # Return
```

# Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label` *(inv. push/pop)*
  - 1) □ Push return address on stack
  - 2) □ Jump to *label*
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address

# Control Flow Example #1

```
000000000400540 <multstore>:
```

```
•  
•  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx)  
•  
•
```

```
000000000400550 <mult2>:
```

```
400550: mov    %rdi, %rax  
•  
•  
400557: retq
```

0x130  
0x128  
0x120

%rsp  
%rip

0x120  
0x400544

instruction pointer - *for fun*

# Control Flow Example #2

```
000000000400540 <multstore>:
```

- 

- 

400544: callq 400550 <mult2>

400549: mov %rax, (%rbx) ←

- 

- 

*mult2 를 invoke 하는지 알 수 있나?*

```
000000000400550 <mult2>:
```

- 

400550: mov %rdi, %rax ←

- 

400557: retq

0x130

0x128

0x120

0x118

0x400549

%rsp

0x118

%rip

0x400550

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

# Control Flow Example #3

```
000000000400540 <multstore>:
```

```
•  
•  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```

```
000000000400550 <mult2>:
```

```
400550: mov    %rdi, %rax  
•  
•  
400557: retq ←
```

0x130

0x128

0x120

0x118

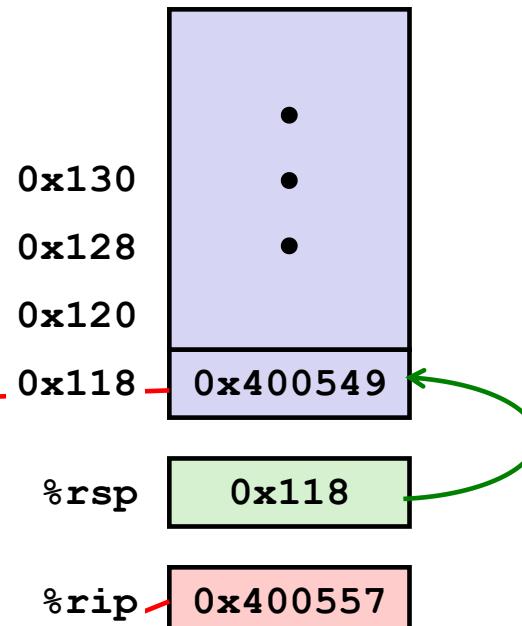
0x400549

%rsp

0x118

%rip

0x400557



# Control Flow Example #4

```
000000000400540 <multstore>:
```

```
•  
•  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←  
•  
•
```

```
000000000400550 <mult2>:
```

```
400550: mov    %rdi, %rax  
•  
•  
400557: retq
```

0x130

0x128

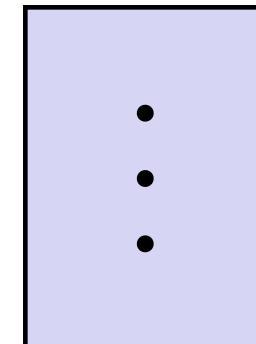
0x120

%rsp

0x120

%rip

0x400549



# Today

## ■ Procedures

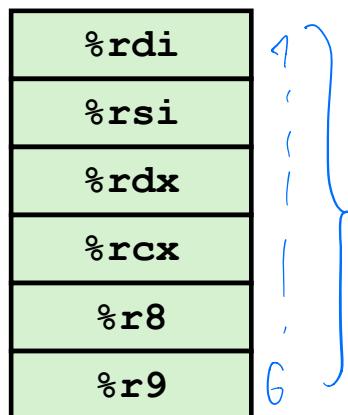
- Mechanisms
- Stack Structure
- Calling Conventions
  - Passing control
  - **Passing data**
  - Managing local data
- Illustrations of Recursion & Pointers

# Procedure Data Flow

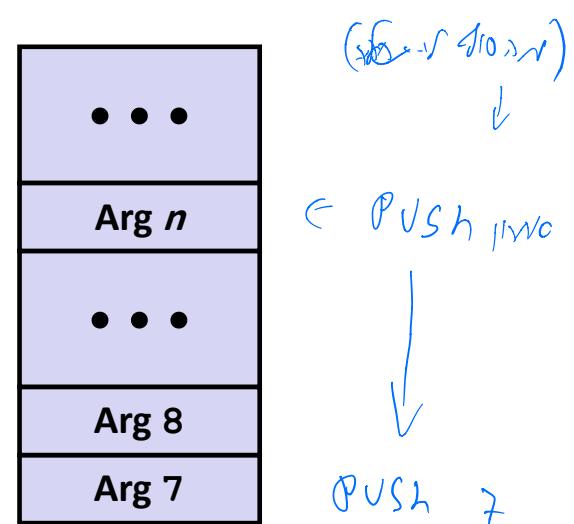
(procedure instruction flow from main to func)

## Registers

- First 6 arguments



## Stack



- Return value



- Only allocate stack space when needed

# Data Flow Examples

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

0000000000400540 <multstore>:

```
# x in %rdi, y in %rsi, dest in %rdx
...
400541: mov    %rdx,%rbx          # Save dest
400544: callq  400550 <mult2>    # mult2(x,y)
# t in %rax
400549: mov    %rax,(%rbx)       # Save at dest
...
```

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

0000000000400550 <mult2>:

```
# a in %rdi, b in %rsi
400550: mov    %rdi,%rax        # a
400553: imul   %rsi,%rax        # a * b
# s in %rax
400557: retq
```

# Call/ret poll

1. call and ret are just like pushq %rip and popq %rip<sub>y</sub>
2. ret instruction does not need any argument<sub>y</sub>

( 823W Stack-& RIP 301 )

# Today

## ■ Procedures

- Mechanisms
- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Stack-Based Languages

## ■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “*Reentrant*”
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

## ■ Stack discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

## ■ Stack allocated in *Frames*

.1 frame w/ 2010 bytes

- state for single procedure instantiation

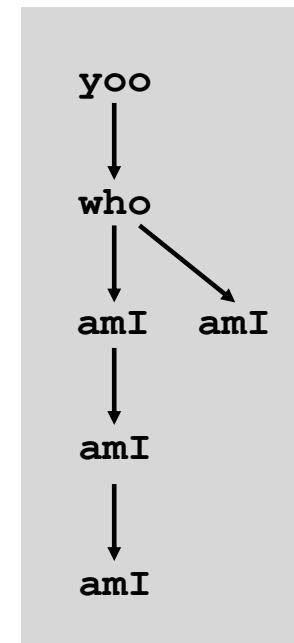
# Call Chain Example

```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```

```
who (...)  
{  
    • • •  
    amI () ;  
    • • •  
    amI () ;  
    • • •  
}
```

```
amI (...)  
{  
    •  
    •  
    amI () ;  
    •  
}
```

## Example Call Chain

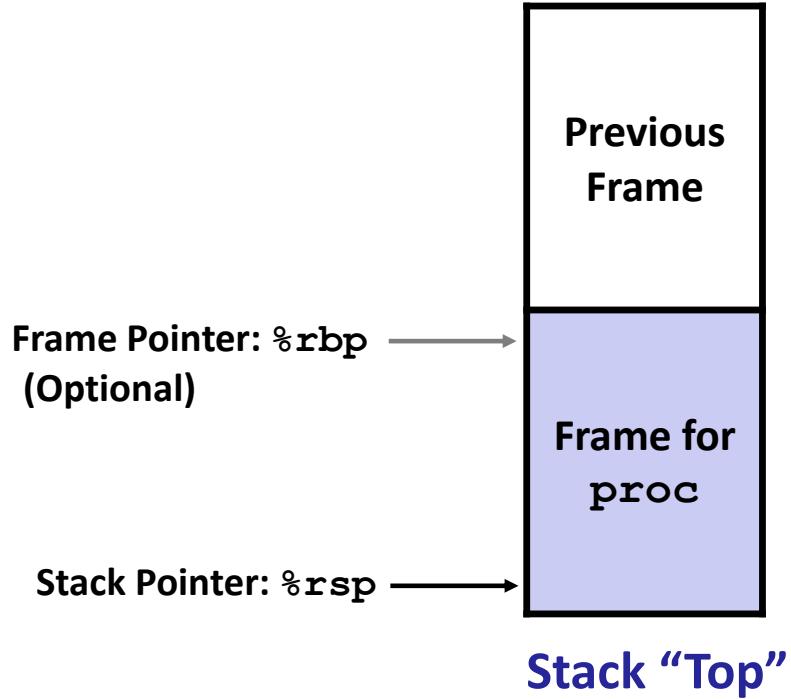


Procedure `amI ()` is recursive

# Stack Frames

## ■ Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)



## ■ Management

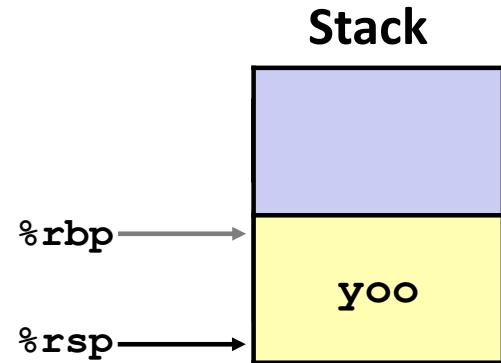
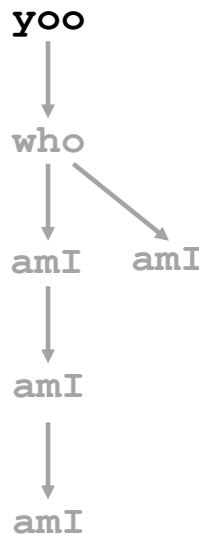
- Space allocated when enter procedure
  - “Set-up” code
  - Includes push by **call** instruction
- Deallocated when return
  - “Finish” code
  - Includes pop by **ret** instruction

*frames' N*

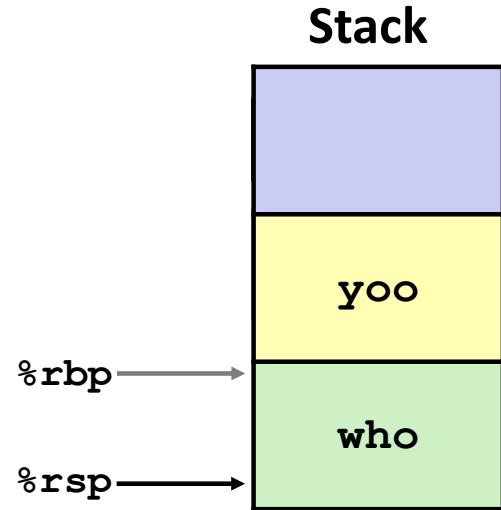
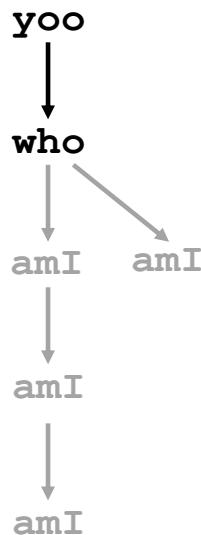
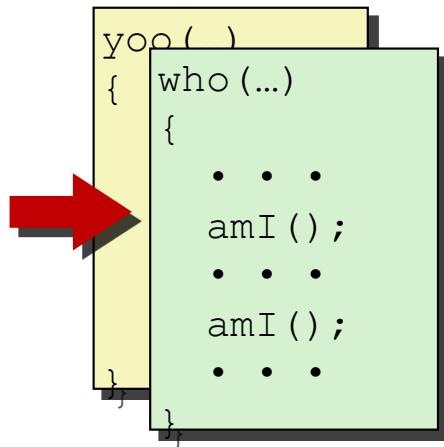
*frame j^N / N*

# Example

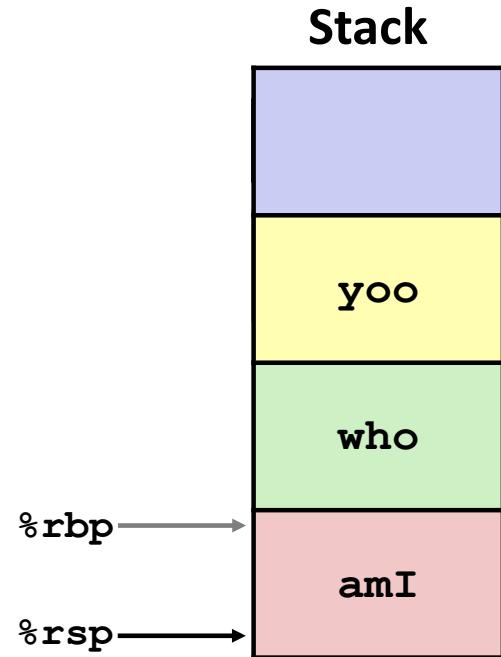
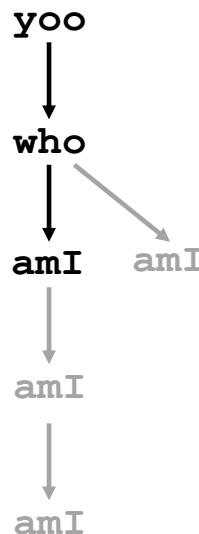
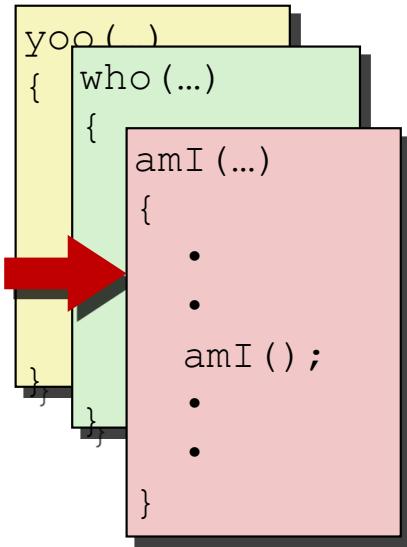
```
yoo (...)  
{  
    .  
    .  
    who ();  
    .  
    .  
}
```



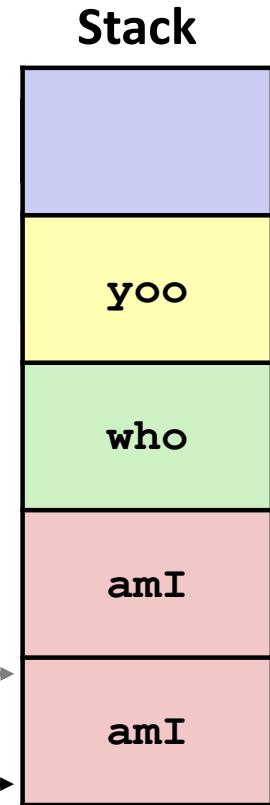
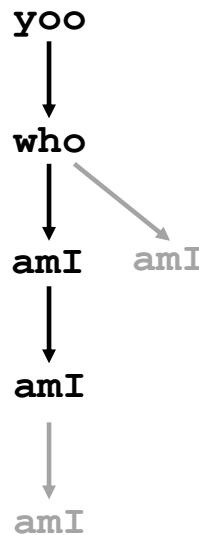
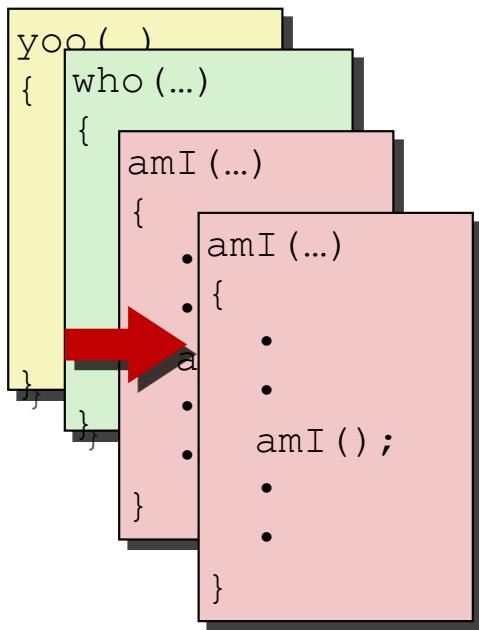
# Example



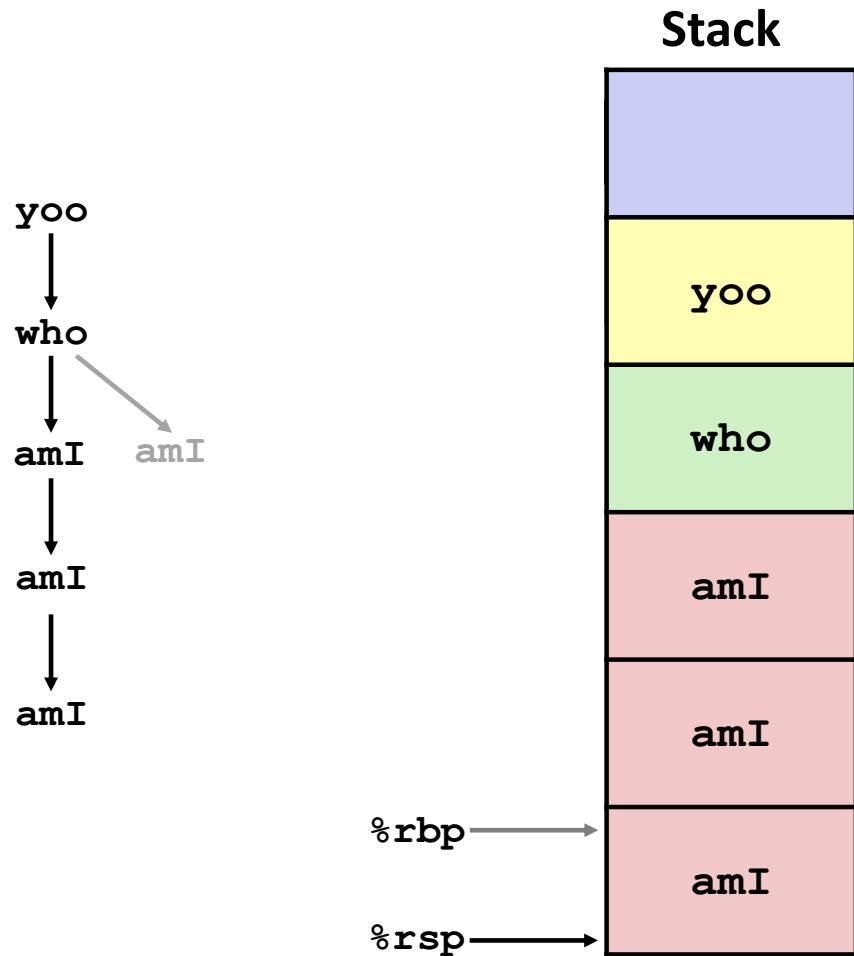
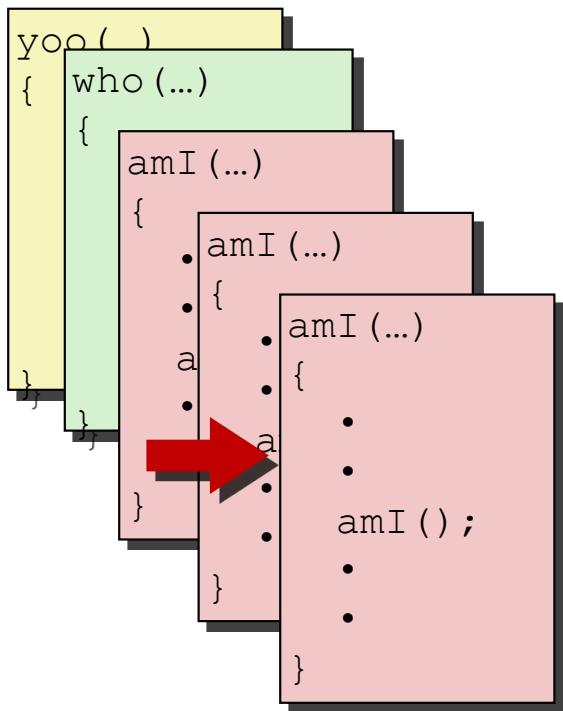
# Example



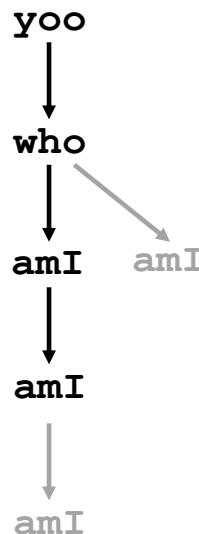
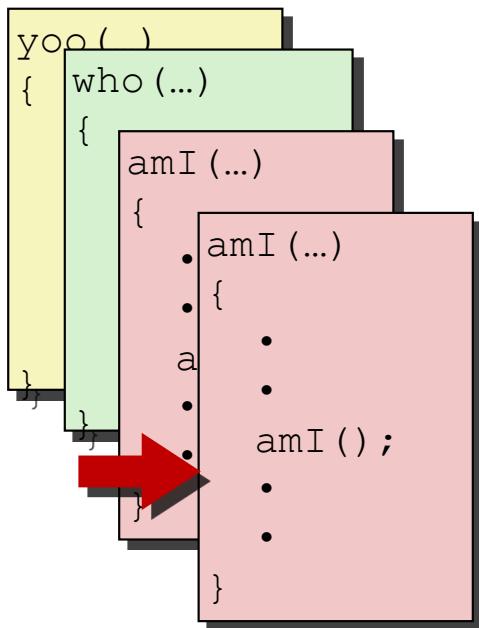
# Example



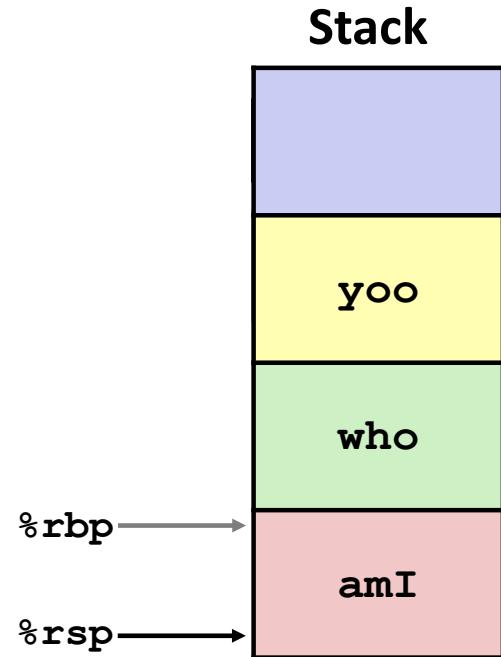
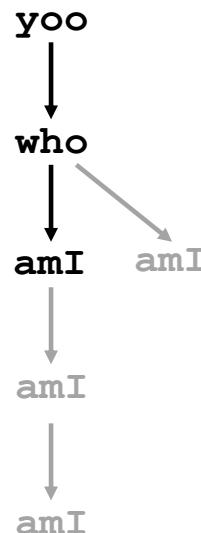
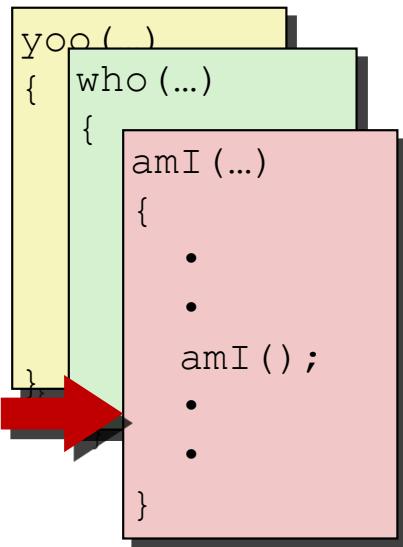
# Example



# Example

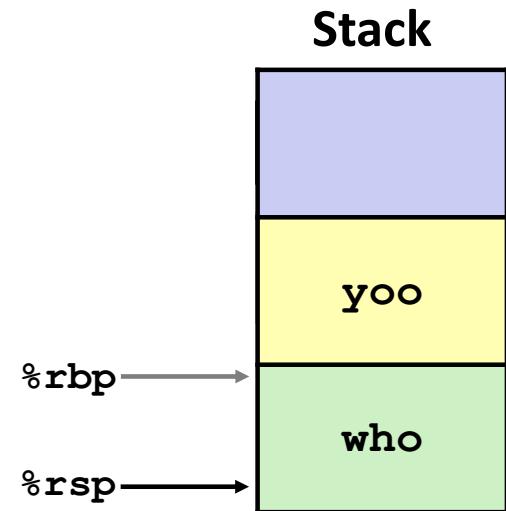
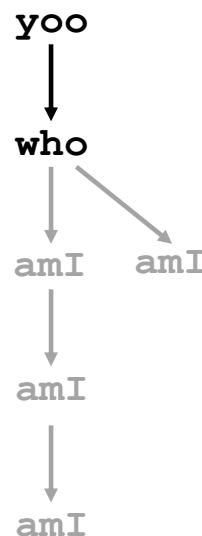


# Example

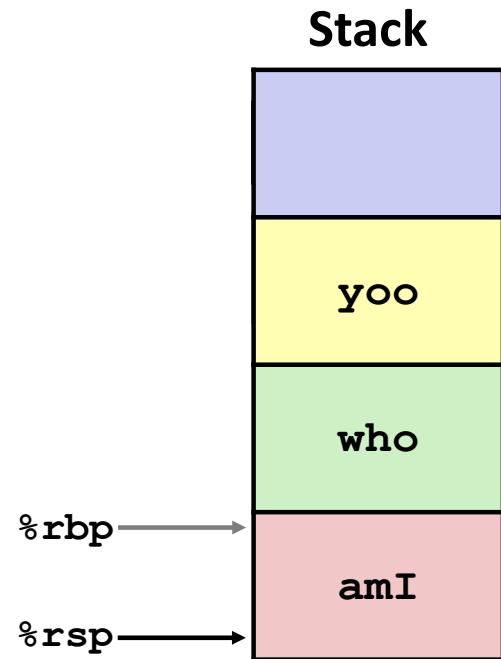
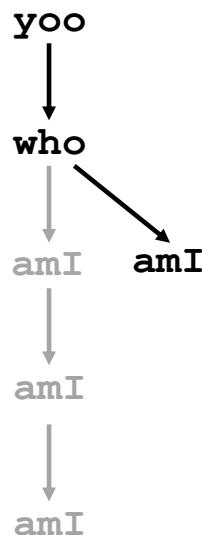
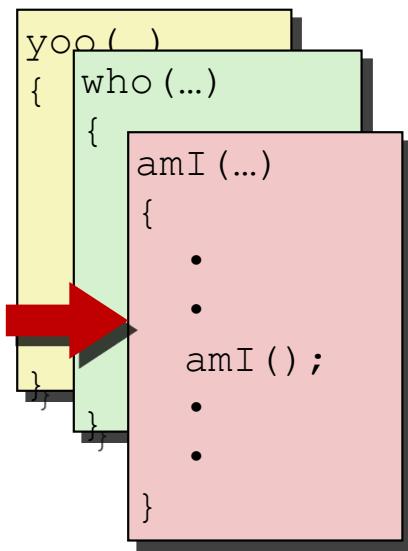


# Example

```
yoo( )  
{  who (...)  
{  
    • • •  
    amI () ;  
    • • •  
    amI () ;  
    • • •  
}
```



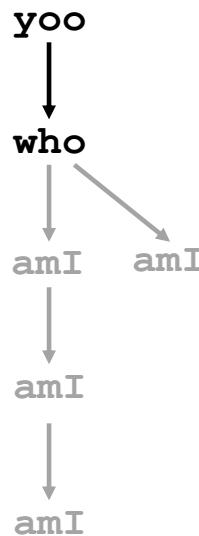
## Example



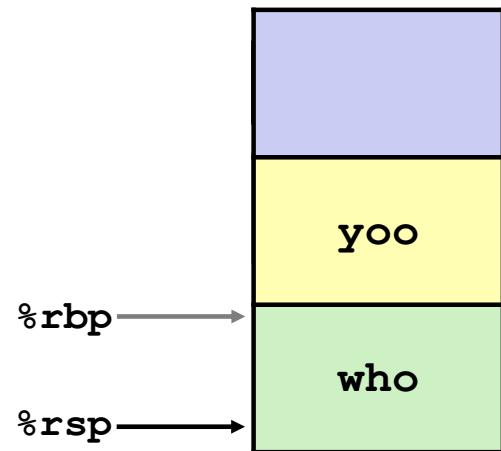
# Example

```
yoo()
{
    who(...)

    {
        . . .
        amI();
        . . .
        amI();
        . . .
    }
}
```

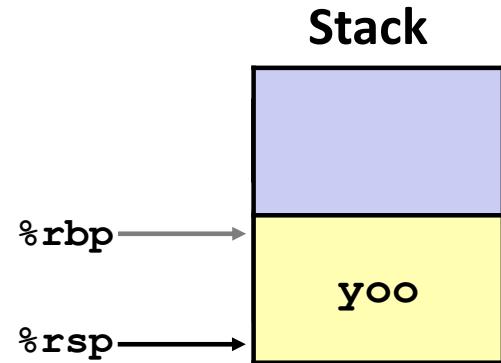
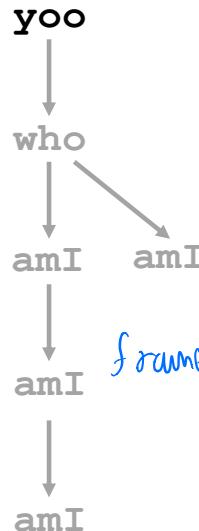


Stack



# Example

```
yoo (...) {  
    •  
    •  
    who () ;  
    •  
    •  
}
```

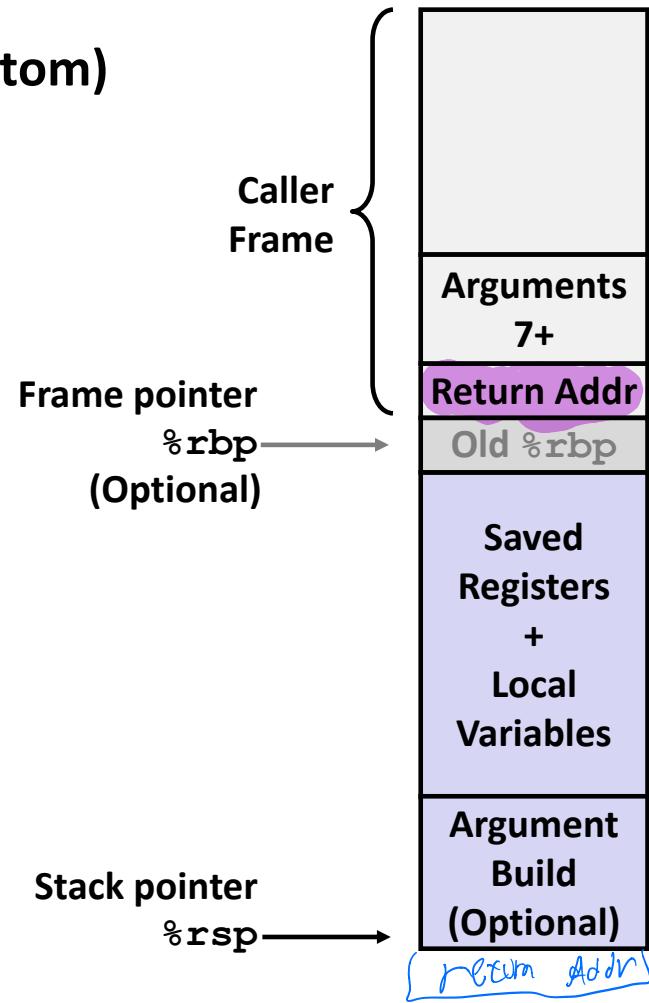


frame of recursive  
function

# x86-64/Linux Stack Frame

## ■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”  
Parameters for function about to call
- Local variables  
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)



## ■ Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call

# Example: `incr`

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

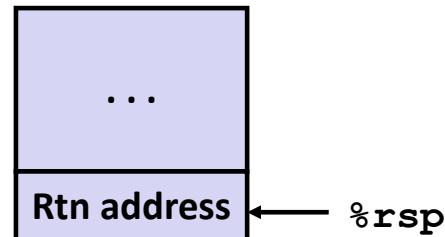
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

| Register | Use(s)                                     |
|----------|--------------------------------------------|
| %rdi     | Argument <code>p</code>                    |
| %rsi     | Argument <code>val</code> , <code>y</code> |
| %rax     | <code>x</code> , Return value              |

# Example: Calling `incr` #1

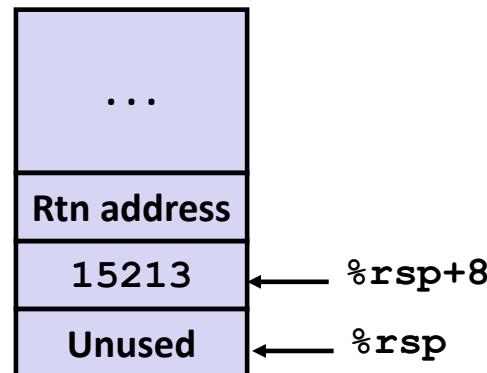
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Resulting Stack Structure

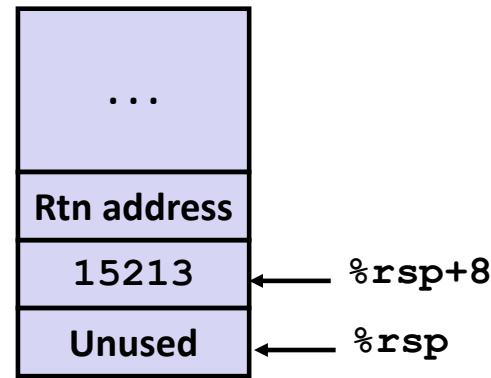


# Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure

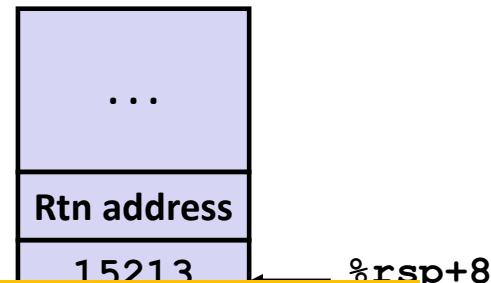


| Register | Use(s) |
|----------|--------|
| %rdi     | &v1    |
| %rsi     | 3000   |

# Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure



Aside 1: `movl $3000, %esi`

`call_`  
`sub_`  
`mov_`

- Note: `movl` -> `%exx` zeros out high order 32 bits.
- Why use `movl` instead of `movq`? 1 byte shorter.

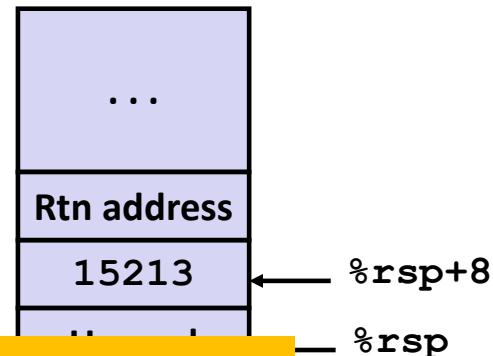
```
movl    $3000, %esi
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

|                   |                      |
|-------------------|----------------------|
| <code>%rdi</code> | <code>&amp;v1</code> |
| <code>%rsi</code> | 3000                 |

# Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

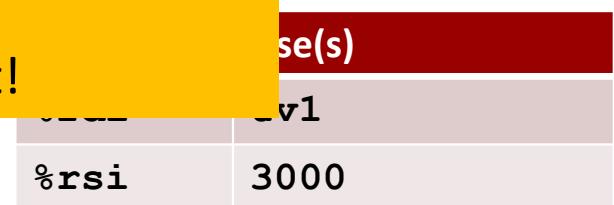
Stack Structure



call\_incr();      Aside 2: `leaq 8(%rsp), %rdi`

- Computes `%rsp+8`
- Actually, used for what it is meant!

```
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

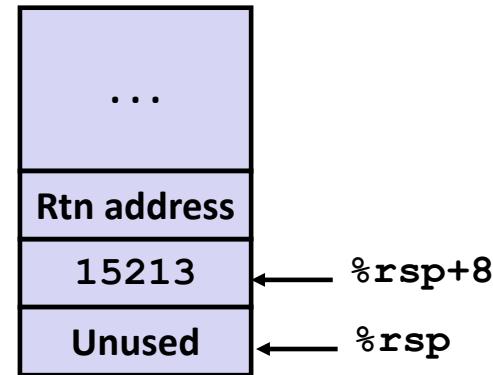


# Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



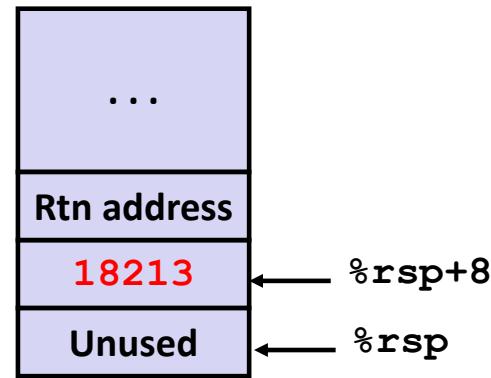
| Register | Use(s) |
|----------|--------|
| %rdi     | &v1    |
| %rsi     | 3000   |

# Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure

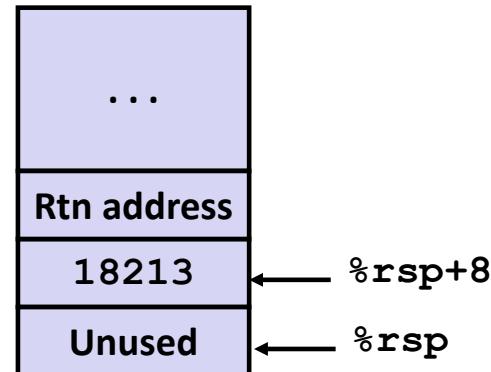


| Register | Use(s) |
|----------|--------|
| %rdi     | &v1    |
| %rsi     | 3000   |

# Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure



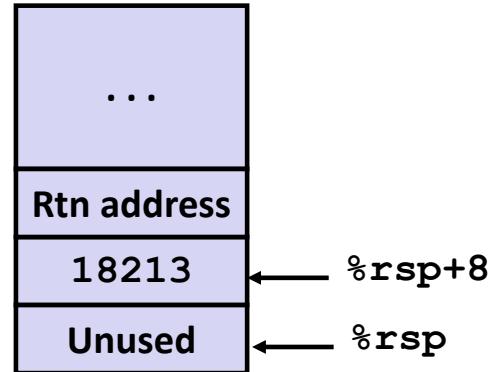
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

| Register | Use(s)       |
|----------|--------------|
| %rax     | Return value |

# Example: Calling `incr` #5a

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure



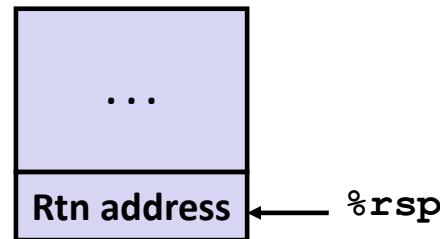
`call_incr:`

```
subq    $16, %rsp
movq    $15213, 8(%rsp)
movl    $3000, %esi
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

(↑↑↑↑↑  
stack ↑↑↑↑↑)

| Register | Use(s)       |
|----------|--------------|
| %rax     | Return value |

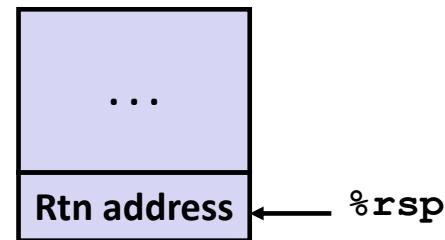
Updated Stack Structure



# Example: Calling `incr` #5b

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

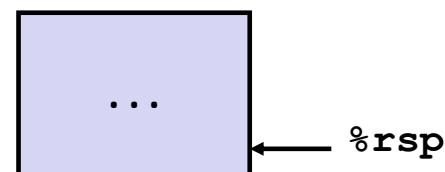
Updated Stack Structure



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

| Register | Use(s)       |
|----------|--------------|
| %rax     | Return value |

Final Stack Structure



(register save/restore)

# Register Saving Conventions

## ■ When procedure `yoo` calls `who`:

- `yoo` is the *caller*
- `who` is the *callee*

## ■ Can register be used for temporary storage?

```
yoo:
```

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

```
who:
```

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

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
  - Need some coordination

# Register Saving Conventions

## ■ When procedure **you** calls **who**:

- **you** is the *caller*
- **who** is the *callee*

register for  
caller uses stack  
callee uses stack

## ■ Can register be used for temporary storage?

## ■ Conventions

### ▪ “*Caller Saved*”

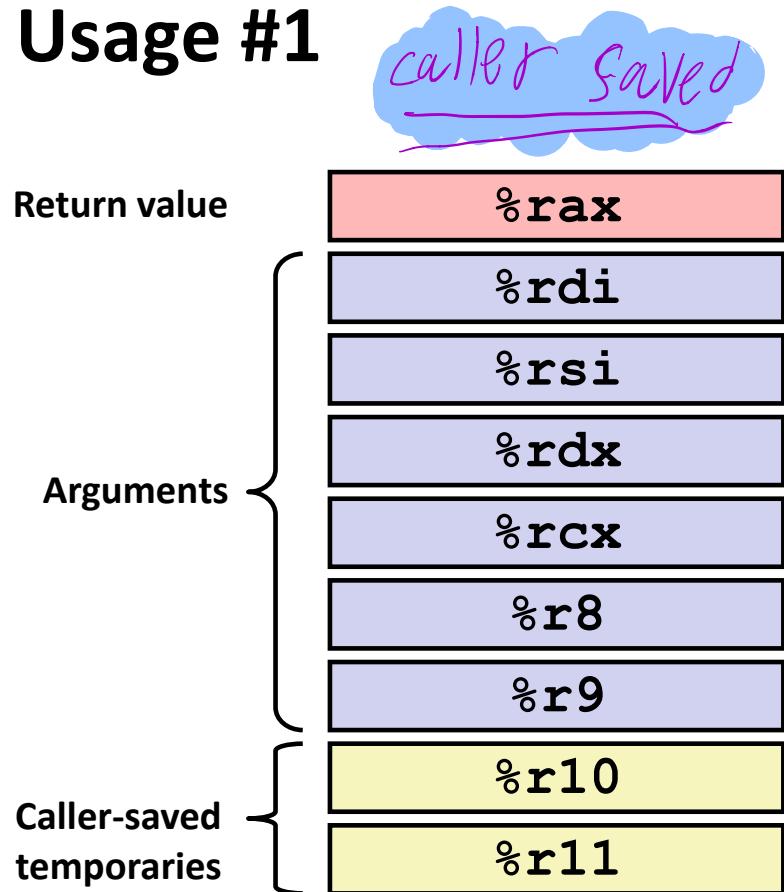
- Caller saves temporary values in its frame before the call

### ▪ “*Callee Saved*”

- Callee saves temporary values in its frame before using
- Callee restores them before returning to caller

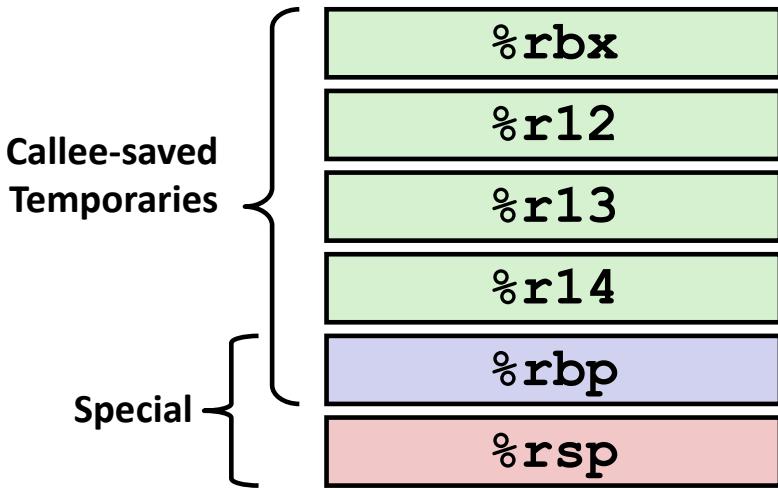
# x86-64 Linux Register Usage #1

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



# x86-64 Linux Register Usage #2

- **%rbx, %r12, %r13, %r14**
  - 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



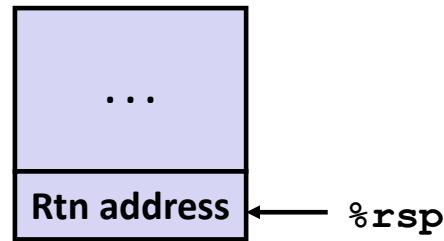
# Callee/caller registers

1. Each function has its own stack frame ↗
2. Callees are required to save all registers before modifying them ↗
3. Only the first 6 arguments to a function are passed in registers ↗
4. Recursion may result in stack growth ?

# Callee-Saved Example #1

```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

Initial Stack Structure

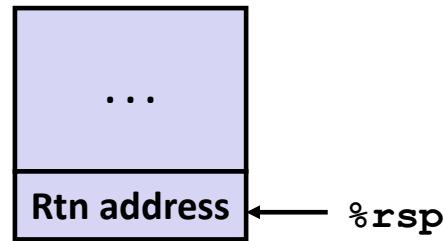


- X comes in register **%rdi**.
- We need **%rdi** for the call to incr.
- Where should be put x, so we can use it after the call to incr?

# Callee-Saved Example #2

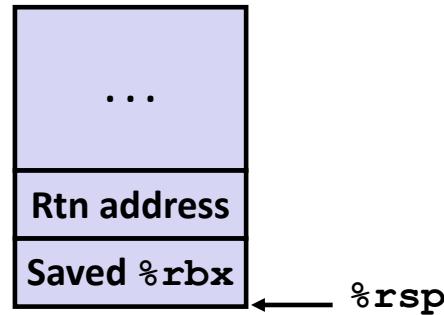
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

Initial Stack Structure



```
call_incr2:  
    pushq  %rbx  (callbb)  
    subq   $16,  %rsp  
    movq   %rdi,  %rbx  
    movq   $15213, 8(%rsp)  
    movl   $3000,  %esi  
    leaq    8(%rsp),  %rdi  
    call    incr  
    addq   %rbx,  %rax  
    addq   $16,  %rsp  
    popq   %rbx  
    ret
```

Resulting Stack Structure

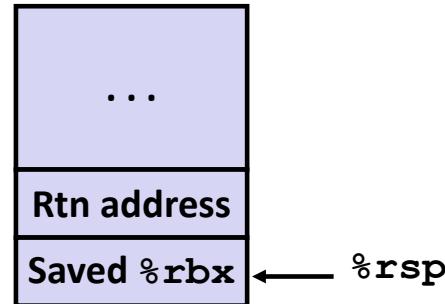


# Callee-Saved Example #3

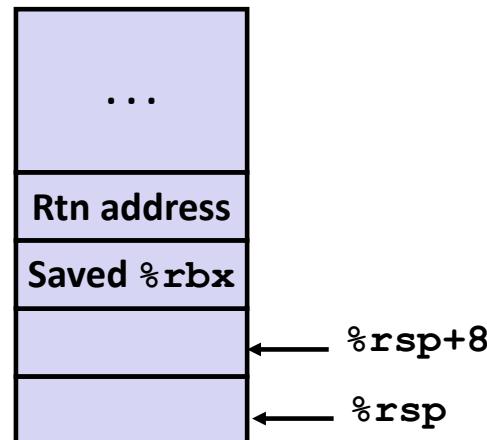
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

Initial Stack Structure



Resulting Stack Structure



# Callee-Saved Example #4

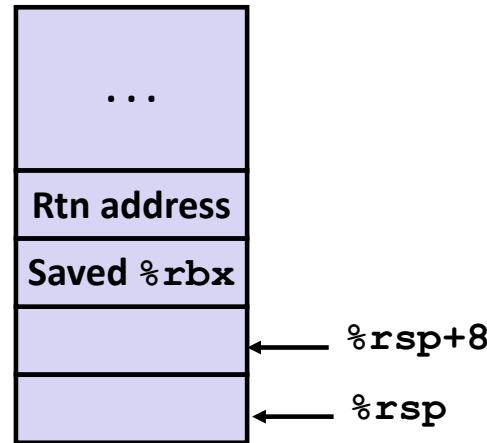
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

(%rbx, rdi, rax)

(%rbx, rdi, rax)

Stack Structure



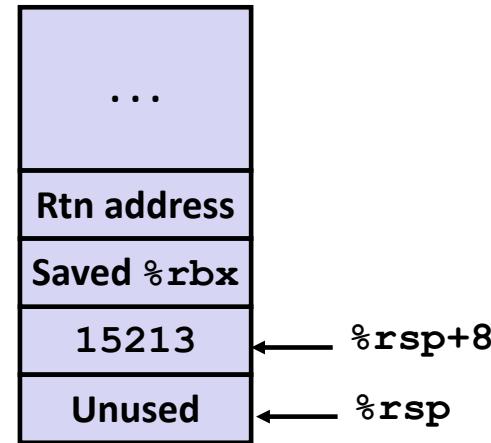
- **x** is saved in **%rbx**,  
a callee saved register

# Callee-Saved Example #5

```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    pushq  %rbx  
    subq   $16, %rsp  
    movq   %rdi, %rbx  
    movq   $15213, 8(%rsp)  
    movl   $3000, %esi  
    leaq   8(%rsp), %rdi  
    call   incr  
    addq   %rbx, %rax  
    addq   $16, %rsp  
    popq   %rbx  
    ret
```

Stack Structure



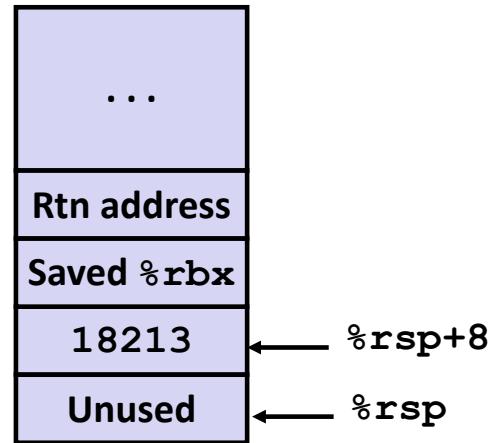
- **x** is saved in **%rbx**,  
a callee saved register

# Callee-Saved Example #6

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

Stack Structure



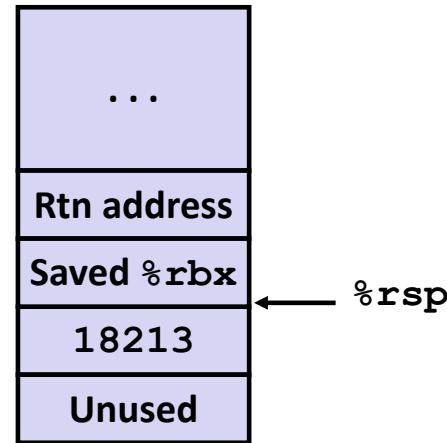
- Upon return from incr:
- **x** is safe in **%rbx**
  - Return result **v2** is in **%rax**
  - Compute **x+v2**

# Callee-Saved Example #7

```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Stack Structure



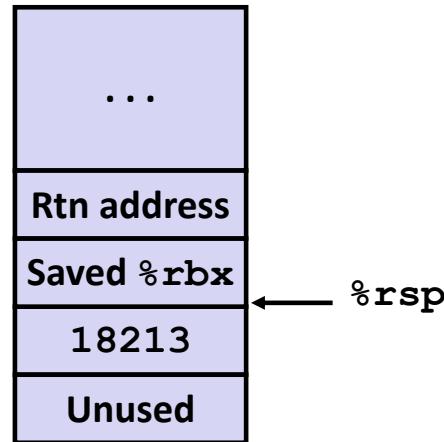
- Return result in `%rax`

# Callee-Saved Example #8

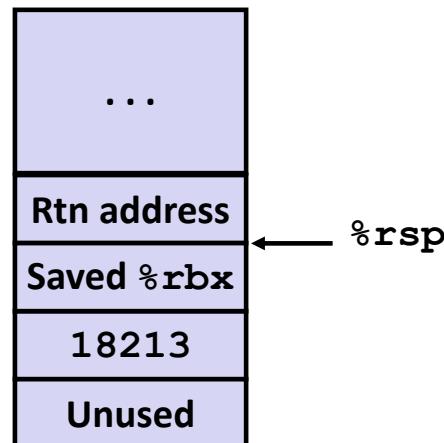
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

Initial Stack Structure



final Stack Structure



# Today

## ■ Procedures

- Mechanisms
- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Recursive Function

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

*(jump equal)*

*(for next level recursive)*

tail:

foo() {  
;  
;  
return foo;  
}

head

foo(){  
foo()  
;  
;  
return;  
}

only 0

pcount\_r:

```
movl $0, %eax
testq %rdi, %rdi
je .L6
pushq %rbx
movq %rdi, %rbx
andl $1, %ebx
shrq %rdi
call pcount_r
addq %rbx, %rax
popq %rbx
```

.L6:

rep; ret

# Recursive Function Terminal Case

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

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

`rep; ret`

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

# Recursive Function Register Save

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

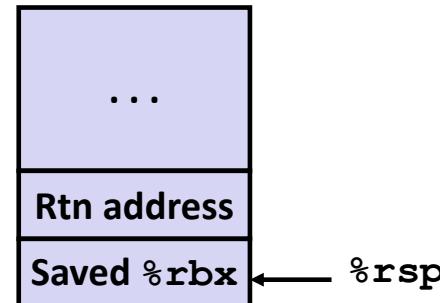
pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbxcall
    movq   ~wel %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

```
rep; ret
```

| Register | Use(s) | Type     |
|----------|--------|----------|
| %rdi     | x      | Argument |



# 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);
}
```

*Recursive update of rdi -> Shift-right operation is shown below*

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

| Register | Use(s) | Type               |
|----------|--------|--------------------|
| %rdi     | x >> 1 | Recursive argument |
| %rbx     | x & 1  | Callee-saved       |

# Recursive Function Call

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

pcount\_r:

|       |                 |
|-------|-----------------|
| movl  | \$0, %eax       |
| testq | %rdi, %rdi      |
| je    | .L6             |
| pushq | %rbx            |
| movq  | %rdi, %rbx      |
| andl  | \$1, %ebx       |
| shrq  | %rdi            |
| call  | <b>pcount_r</b> |
| addq  | %rbx, %rax      |
| popq  | %rbx            |

.L6:

**rep; ret**

| Register | Use(s)                         | Type         |
|----------|--------------------------------|--------------|
| %rbx     | x & 1                          | Callee-saved |
| %rax     | Recursive call<br>return value |              |

# Recursive Function Result

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

pcount\_r:

|       |            |
|-------|------------|
| movl  | \$0, %eax  |
| testq | %rdi, %rdi |
| je    | .L6        |
| pushq | %rbx       |
| movq  | %rdi, %rbx |
| andl  | \$1, %ebx  |
| shrq  | %rdi       |
| call  | pcount_r   |
| addq  | %rbx, %rax |
| popq  | %rbx       |

.L6:  
rep; ret

(rdi  
rpbrr)

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

# 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 |

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret(%rdx %rax)  
↓↓↓↓↓



← %rsp

# Observations About Recursion

## ■ Handled Without Special Consideration

פונקציית פון פון פון  
הCALL מושך מושך מושך  
הRET מושך מושך מושך

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

## ■ Also works for mutual recursion

- P calls Q; Q calls P

# x86-64 Procedure Summary

## ■ Important Points

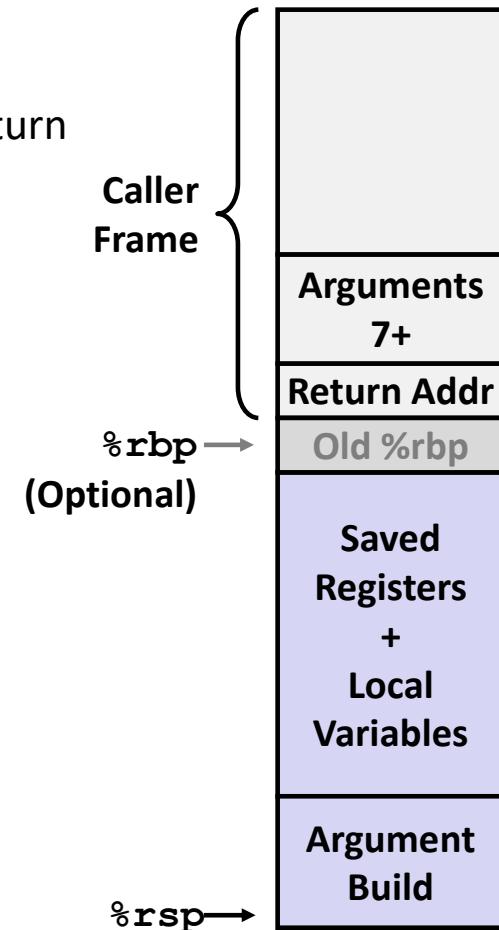
- Stack is the right data structure for procedure call/return
  - If P calls Q, then Q returns before P

## ■ Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in `%rax`

## ■ Pointers are addresses of values

- On stack or global



# Small Exercise

```

long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

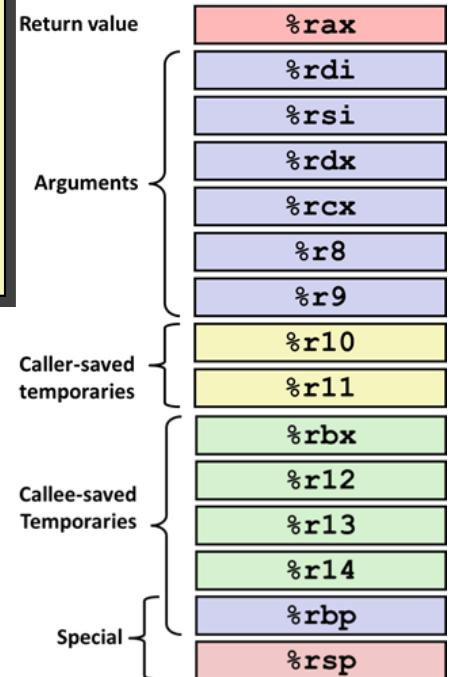
long add10(long a0, long a1, long a2, long a3, long a4, long a5,
           long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4) +
           add5(a5, a6, a7, a8, a9);
}

```

■ Where are  $a_0, \dots, a_9$  passed?  
**rdi, rsi, rdx, rcx, r8, r9, stack**

■ Where are  $b_0, \dots, b_4$  passed?  
**rdi, rsi, rdx, rcx, r8**

■ Which registers do we need to save?  
**rbx, rbp, r9 (during first call to add5)**

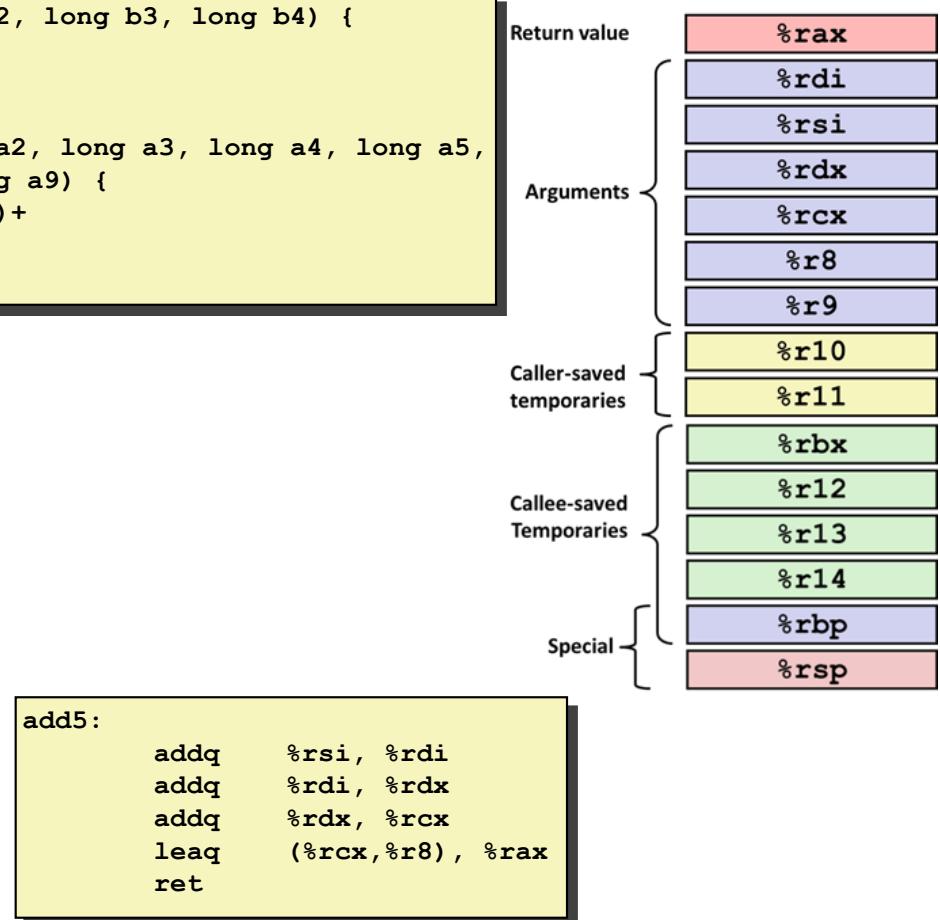


# Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5,
           long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4) +
           add5(a5, a6, a7, a8, a9);
}
```

```
add10:
    pushq  %rbp
    pushq  %rbx
    movq   %r9, %rbp
    call   add5
    movq   %rax, %rbx
    movq   48(%rsp), %r8
    movq   40(%rsp), %rcx
    movq   32(%rsp), %rdx
    movq   24(%rsp), %rsi
    movq   %rbp, %rdi
    call   add5
    addq   %rbx, %rax
    popq   %rbx
    popq   %rbp
    ret
```



$$\therefore \mu \leq 0$$

• 'N' is the first letter of the word 'NAND3' (1)

የፍትህ አገልግሎት የፍትህ ስነዎች

(circle  $\cup$   $S^1$ )

לעתה נוכיח:  $\lim_{n \rightarrow \infty} \int_0^1 f_n(x) dx = \int_0^1 f(x) dx$

۱۷۰۰ میلادی میان گزینه هایی که

לעתים מושג מינימום בפונקציית האנרגיה (c)

•  $\text{R} \approx 10\sqrt{2}$

— תְּמִימָנֶה בְּרִית מִצְרַיִם וְעֵמֶק יָמִינָה — תְּמִימָנֶה בְּרִית מִצְרַיִם וְעֵמֶק יָמִינָה (2)

(n) vs p n) journal

"Pd" k( )

11

יצאנו מפונקיה push עם נסס push prev res (יכלול בפונקיה push)

For the first major pop series, the PSP

הנתקן בראון

join several frames together

01)  $\int_{-1}^1 \sqrt{1-x^2} dx$  für die Stammfunktionen und die Fläche (C)

የኢትዮጵያውያንድ ስራውን አገልግሎት የሚከተሉት በቻ የሚከተሉት በቻ የሚከተሉት በቻ

(۲۷)  $\int x^2 \sin x dx$

(push)

. या ना योगी

(1) הנורווגיה: אקלים ממוזג ים-תיכוני.

فَدَعَاهُمْ (٦٨)

۱۰۰۰ Word + الگویی و مفهومی

השורה התחתונה (bottom row) מוגדרת כ

(POP) POP (top row): נזקן מוכנה (bottom row)

השורה התיכונה (middle row) מוגדרת כ

(RDI, RSI, RBX, RCX, R8, R9)

RAX: הולך ו诙ילן RDI ב-15fe (ה)

בשורה תיכונה מוגדרת (frame) ret → R10 (ה)

RAX-ה יושם בו גורם דמי אוניברסיטאי (ה)

השורה עליונה (top row) מוגדרת כ

frame RDI : הולך ו诙ילן R10 (ה)

השורה עליונה מוגדרת כ

frame RDI : הולך ו诙ילן R10 (ה)

השורה עליונה מוגדרת כ

frame RDI : הולך ו诙ילן R10 (ה)

השורה עליונה מוגדרת כ

frame RDI : הולך ו诙ילן R10 (ה)

RAX, RDI-R9,R10,R11,R12,R13,R14,R15,R16,R17,R18,R19,R20,R21,R22,R23,R24,R25,R26,R27,R28,R29,R30,R31,R32,R33,R34,R35,R36,R37,R38,R39,R40,R41,R42,R43,R44,R45,R46,R47,R48,R49,R50,R51,R52,R53,R54,R55,R56,R57,R58,R59,R60,R61,R62,R63,R64,R65,R66,R67,R68,R69,R70,R71,R72,R73,R74,R75,R76,R77,R78,R79,R80,R81,R82,R83,R84,R85,R86,R87,R88,R89,R89,R90,R91,R92,R93,R94,R95,R96,R97,R98,R99,R99,R100,R101,R102,R103,R104,R105,R106,R107,R108,R109,R109,R110,R111,R112,R113,R114,R115,R116,R117,R118,R119,R119,R120,R121,R122,R123,R124,R125,R126,R127,R128,R129,R129,R130,R131,R132,R133,R134,R135,R136,R137,R138,R139,R139,R140,R141,R142,R143,R144,R145,R146,R147,R148,R149,R149,R150,R151,R152,R153,R154,R155,R156,R157,R158,R159,R159,R160,R161,R162,R163,R164,R165,R166,R167,R168,R169,R169,R170,R171,R172,R173,R174,R175,R176,R177,R178,R179,R179,R180,R181,R182,R183,R184,R185,R186,R187,R188,R189,R189,R190,R191,R192,R193,R194,R195,R196,R197,R198,R199,R199,R200,R201,R202,R203,R204,R205,R206,R207,R208,R209,R209,R210,R211,R212,R213,R214,R215,R216,R217,R218,R219,R219,R220,R221,R222,R223,R224,R225,R226,R227,R228,R229,R229,R230,R231,R232,R233,R234,R235,R236,R237,R238,R239,R239,R240,R241,R242,R243,R244,R245,R246,R247,R248,R249,R249,R250,R251,R252,R253,R254,R255,R256,R257,R258,R259,R259,R260,R261,R262,R263,R264,R265,R266,R267,R268,R269,R269,R270,R271,R272,R273,R274,R275,R276,R277,R278,R279,R279,R280,R281,R282,R283,R284,R285,R286,R287,R288,R289,R289,R290,R291,R292,R293,R294,R295,R296,R297,R298,R299,R299,R300,R301,R302,R303,R304,R305,R306,R307,R308,R309,R309,R310,R311,R312,R313,R314,R315,R316,R317,R318,R319,R319,R320,R321,R322,R323,R324,R325,R326,R327,R328,R329,R329,R330,R331,R332,R333,R334,R335,R336,R337,R338,R339,R339,R340,R341,R342,R343,R344,R345,R346,R347,R348,R349,R349,R350,R351,R352,R353,R354,R355,R356,R357,R358,R359,R359,R360,R361,R362,R363,R364,R365,R366,R367,R368,R369,R369,R370,R371,R372,R373,R374,R375,R376,R377,R378,R379,R379,R380,R381,R382,R383,R384,R385,R386,R387,R388,R389,R389,R390,R391,R392,R393,R394,R395,R396,R397,R398,R398,R399,R399,R400,R399,R401,R402,R403,R404,R405,R406,R407,R408,R409,R409,R410,R411,R412,R413,R414,R415,R416,R417,R418,R419,R419,R420,R421,R422,R423,R424,R425,R426,R427,R428,R429,R429,R430,R431,R432,R433,R434,R435,R436,R437,R438,R439,R439,R440,R441,R442,R443,R444,R445,R446,R447,R448,R449,R449,R450,R451,R452,R453,R454,R455,R456,R457,R458,R459,R459,R460,R461,R462,R463,R464,R465,R466,R467,R468,R469,R469,R470,R471,R472,R473,R474,R475,R476,R477,R478,R479,R479,R480,R481,R482,R483,R484,R485,R486,R487,R488,R489,R489,R490,R491,R492,R493,R494,R495,R496,R497,R498,R498,R499,R499,R500,R499,R501,R502,R503,R504,R505,R506,R507,R508,R509,R509,R510,R511,R512,R513,R514,R515,R516,R517,R518,R519,R519,R520,R521,R522,R523,R524,R525,R526,R527,R528,R529,R529,R530,R531,R532,R533,R534,R535,R536,R537,R538,R539,R539,R540,R541,R542,R543,R544,R545,R546,R547,R548,R549,R549,R550,R551,R552,R553,R554,R555,R556,R557,R558,R559,R559,R560,R561,R562,R563,R564,R565,R566,R567,R568,R569,R569,R570,R571,R572,R573,R574,R575,R576,R577,R578,R579,R579,R580,R581,R582,R583,R584,R585,R586,R587,R588,R589,R589,R590,R591,R592,R593,R594,R595,R596,R597,R598,R598,R599,R599,R600,R599,R601,R602,R603,R604,R605,R606,R607,R608,R609,R609,R610,R611,R612,R613,R614,R615,R616,R617,R618,R619,R619,R620,R621,R622,R623,R624,R625,R626,R627,R628,R629,R629,R630,R631,R632,R633,R634,R635,R636,R637,R638,R639,R639,R640,R641,R642,R643,R644,R645,R646,R647,R648,R649,R649,R650,R651,R652,R653,R654,R655,R656,R657,R658,R659,R659,R660,R661,R662,R663,R664,R665,R666,R667,R668,R669,R669,R670,R671,R672,R673,R674,R675,R676,R677,R678,R679,R679,R680,R681,R682,R683,R684,R685,R686,R687,R688,R689,R689,R690,R691,R692,R693,R694,R695,R696,R697,R698,R698,R699,R699,R700,R699,R701,R702,R703,R704,R705,R706,R707,R708,R709,R709,R710,R711,R712,R713,R714,R715,R716,R717,R718,R719,R719,R720,R721,R722,R723,R724,R725,R726,R727,R728,R729,R729,R730,R731,R732,R733,R734,R735,R736,R737,R738,R739,R739,R740,R741,R742,R743,R744,R745,R746,R747,R748,R749,R749,R750,R751,R752,R753,R754,R755,R756,R757,R758,R759,R759,R760,R761,R762,R763,R764,R765,R766,R767,R768,R769,R769,R770,R771,R772,R773,R774,R775,R776,R777,R778,R779,R779,R780,R781,R782,R783,R784,R785,R786,R787,R788,R789,R789,R790,R791,R792,R793,R794,R795,R796,R797,R798,R798,R799,R799,R800,R799,R801,R802,R803,R804,R805,R806,R807,R808,R809,R809,R810,R811,R812,R813,R814,R815,R816,R817,R818,R819,R819,R820,R821,R822,R823,R824,R825,R826,R827,R828,R829,R829,R830,R831,R832,R833,R834,R835,R836,R837,R838,R839,R839,R840,R841,R842,R843,R844,R845,R846,R847,R848,R849,R849,R850,R851,R852,R853,R854,R855,R856,R857,R858,R859,R859,R860,R861,R862,R863,R864,R865,R866,R867,R868,R869,R869,R870,R871,R872,R873,R874,R875,R876,R877,R878,R879,R879,R880,R881,R882,R883,R884,R885,R886,R887,R888,R889,R889,R890,R891,R892,R893,R894,R895,R896,R897,R898,R898,R899,R899,R900,R899,R901,R902,R903,R904,R905,R906,R907,R908,R909,R909,R910,R911,R912,R913,R914,R915,R916,R917,R918,R919,R919,R920,R921,R922,R923,R924,R925,R926,R927,R928,R929,R929,R930,R931,R932,R933,R934,R935,R936,R937,R938,R939,R939,R940,R941,R942,R943,R944,R945,R946,R947,R948,R949,R949,R950,R951,R952,R953,R954,R955,R956,R957,R958,R959,R959,R960,R961,R962,R963,R964,R965,R966,R967,R968,R969,R969,R970,R971,R972,R973,R974,R975,R976,R977,R978,R979,R979,R980,R981,R982,R983,R984,R985,R986,R987,R988,R988,R989,R989,R990,R989,R991,R992,R993,R994,R995,R996,R997,R998,R998,R999,R999,R1000,R999,R1001,R1002,R1003,R1004,R1005,R1006,R1007,R1008,R1009,R1009,R1010,R1011,R1012,R1013,R1014,R1015,R1016,R1017,R1018,R1019,R1019,R1020,R1021,R1022,R1023,R1024,R1025,R1026,R1027,R1028,R1029,R1029,R1030,R1031,R1032,R1033,R1034,R1035,R1036,R1037,R1038,R1039,R1039,R1040,R1041,R1042,R1043,R1044,R1045,R1046,R1047,R1048,R1049,R1049,R1050,R1051,R1052,R1053,R1054,R1055,R1056,R1057,R1058,R1059,R1059,R1060,R1061,R1062,R1063,R1064,R1065,R1066,R1067,R1068,R1069,R1069,R1070,R1071,R1072,R1073,R1074,R1075,R1076,R1077,R1078,R1079,R1079,R1080,R1081,R1082,R1083,R1084,R1085,R1086,R1087,R1088,R1089,R1089,R1090,R1091,R1092,R1093,R1094,R1095,R1096,R1097,R1098,R1098,R1099,R1099,R1100,R1099,R1101,R1102,R1103,R1104,R1105,R1106,R1107,R1108,R1109,R1109,R1110,R1111,R1112,R1113,R1114,R1115,R1116,R1117,R1118,R1119,R1119,R1120,R1121,R1122,R1123,R1124,R1125,R1126,R1127,R1128,R1129,R1129,R1130,R1131,R1132,R1133,R1134,R1135,R1136,R1137,R1138,R1139,R1139,R1140,R1141,R1142,R1143,R1144,R1145,R1146,R1147,R1148,R1149,R1149,R1150,R1151,R1152,R1153,R1154,R1155,R1156,R1157,R1158,R1159,R1159,R1160,R1161,R1162,R1163,R1164,R1165,R1166,R1167,R1168,R1169,R1169,R1170,R1171,R1172,R1173,R1174,R1175,R1176,R1177,R1178,R1179,R1179,R1180,R1181,R1182,R1183,R1184,R1185,R1186,R1187,R1188,R1189,R1189,R1190,R1191,R1192,R1193,R1194,R1195,R1196,R1197,R1198,R1198,R1199,R1199,R1200,R1199,R1201,R1202,R1203,R1204,R1205,R1206,R1207,R1208,R1209,R1209,R1210,R1211,R1212,R1213,R1214,R1215,R1216,R1217,R1218,R1219,R1219,R1220,R1221,R1222,R1223,R1224,R1225,R1226,R1227,R1228,R1229,R1229,R1230,R1231,R1232,R1233,R1234,R1235,R1236,R1237,R1238,R1239,R1239,R1240,R1241,R1242,R1243,R1244,R1245,R1246,R1247,R1248,R1249,R1249,R1250,R1251,R1252,R1253,R1254,R1255,R1256,R1257,R1258,R1259,R1259,R1260,R1261,R1262,R1263,R1264,R1265,R1266,R1267,R1268,R1269,R1269,R1270,R1271,R1272,R1273,R1274,R1275,R1276,R1277,R1278,R1279,R1279,R1280,R1281,R1282,R1283,R1284,R1285,R1286,R1287,R1288,R1289,R1289,R1290,R1291,R1292,R1293,R1294,R1295,R1296,R1297,R1298,R1298,R1299,R1299,R1300,R1299,R1301,R1302,R1303,R1304,R1305,R1306,R1307,R1308,R1309,R1309,R1310,R1311,R1312,R1313,R1314,R1315,R1316,R1317,R1318,R1319,R1319,R1320,R1321,R1322,R1323,R1324,R1325,R1326,R1327,R1328,R1329,R1329,R1330,R1331,R1332,R1333,R1334,R1335,R1336,R1337,R1338,R1339,R1339,R1340,R1341,R1342,R1343,R1344,R1345,R1346,R1347,R1348,R1349,R1349,R1350,R1351,R1352,R1353,R1354,R1355,R1356,R1357,R1358,R1359,R1359,R1360,R1361,R1362,R1363,R1364,R1365,R1366,R1367,R1368,R1369,R1369,R1370,R1371,R1372,R1373,R1374,R1375,R1376,R1377,R1378,R1379,R1379,R1380,R1381,R1382,R1383,R1384,R1385,R1386,R1387,R1388,R1389,R1389,R1390,R1391,R1392,R1393,R1394,R1395,R1396,R1397,R1398,R1398,R1399,R1399,R1400,R1399,R1401,R1402,R1403,R1404,R1405,R1406,R1407,R1408,R1409,R1409,R1410,R1411,R1412,R1413,R1414,R1415,R1416,R1417,R1418,R1419,R1419,R1420,R1421,R1422,R1423,R1424,R1425,R1426,R1427,R1428,R1429,R1429,R1430,R1431,R1432,R1433,R1434,R1435,R1436,R1437,R1438,R1439,R1439,R1440,R1441,R1442,R1443,R1444,R1445,R1446,R1447,R1448,R1449,R1449,R1450,R1451,R1452,R1453,R1454,R1455,R1456,R1457,R1458,R1459,R1459,R1460,R1461,R1462,R1463,R1464,R1465,R1466,R1467,R1468,R1469,R1469,R1470,R1471,R1472,R1473,R1474,R1475,R1476,R1477,R1478,R1479,R1479,R1480,R1481,R1482,R1483,R1484,R1485,R1486,R1487,R1488,R1489,R1489,R1490,R1491,R1492,R1493,R1494,R1495,R1496,R1497,R1498,R1498,R1499,R1499,R1500,R1499,R1501,R1502,R1503,R1504,R1505,R1506,R1507,R1508,R1509,R1509,R1510,R1511,R1512,R1513,R1514,R1515,R1516,R1517,R1518,R1519,R1519,R1520,R1521,R1522,R1523,R1524,R1525,R1526,R1527,R1528,R1529,R1529,R1530,R1531,R1532,R1533,R1534,R1535,R1536,R1537,R1538,R1539,R1539,R1540,R1541,R1542,R1543,R1544,R1545,R1546,R1547,R1548,R1549,R1549,R1550,R1551,R1552,R1553,R1554,R1555,R1556,R1557,R1558,R1559,R1559,R1560,R1561,R1562,R1563,R1564,R1565,R1566,R1567,R1568,R1569,R1569,R1570,R1571,R1572,R1573,R1574,R1575,R1576,R1577,R1578,R1579,R1579,R1580,R1581,R1582,R1583,R1584,R1585,R1586,R1587,R1588,R1589,R1589,R1590,R1591,R1592,R1593,R1594,R1595,R1596,R1597,R1598,R1598,R1599,R1599,R1600,R1599,R1601,R1602,R1603,R1604,R1605,R1606,R1607,R1608,R1609,R1609,R1610,R1611,R1612,R1613,R1614,R1615,R1616,R1617,R1618,R1619,R1619,R1620,R1621,R1622,R1623,R1624,R1625,R1626,R1627,R1628,R1629,R1629,R1630,R1631,R1632,R1633,R1634,R1635,R1636,R1637,R1638,R1639,R1639,R1640,R1641,R1642,R1643,R1644,R1645,R1646,R1647,R1648,R1649,R1649,R1650,R1651,R1652,R1653,R1654,R1655,R1656,R1657,R1658,R1659,R1659,R1660,R1661,R1662,R1663,R1664,R1665,R1666,R1667,R1668,R1669,R1669,R1670,R1671,R1672,R1673,R1674,R1675,R1676,R1677,R1678,R1679,R1679,R1680,R1681,R1682,R1683,R1684,R1685,R1686,R1687,R1688,R1689,R1689,R1690,R1691,R1692,R1693,R1694,R1695,R1696,R1697,R1698,R1698,R1699,R1699,R1700,R1699,R1701,R1702,R1703,R1704,R1705,R1706,R1707,R1708,R1709,R1709,R1710,R1711,R1712,R1713,R1714,R1715,R1716,R1717,R1718,R1719,R1719,R1720,R1721,R1722,R1723,R1724,R1725,R1726,R1727,R1728,R1729,R1729,R1730,R1731,R1732,R1733,R1734,R1735,R1736,R1737,R1738,R1739,R1739,R1740,R1741,R1742,R1743,R1744,R1745,R1746,R1747,R1748,R1749,R1749,R1750,R1751,R1752,R1753,R1754,R1755,R1756,R1757,R1758,R1759,R1759,R1760,R1761,R1762,R1763,R1764,R1765,R1766,R1767,R1768,R1769,R1769,R1770,R1771,R1772,R1773,R1774,R1775,R1776,R1777,R1778,R1779,R1779,R1780,R1781,R1782,R1783,R1784,R1785,R1786,R1787,R1788,R1789,R1789,R1790,R1791,R1792,R1793,R1794,R1795,R1796,R1797,R1798,R1798,R1799,R1799,R1800,R1799,R1801,R1802,R1803,R1804,R1805,R1806,R1807,R1808,R1809,R1809,R1810,R1811,R1812,R1813,R1814,R1815,R1816,R1817,R1818,R1819,R1819,R1820,R1821,R1822,R1823,R1824,R1825,R1826,R1827,R1828,R1829,R1829,R1830,R1831,R1832,R1833,R1834,R1835,R1836,R1837,R1838,R1839,R1839,R1840,R1841,R1842,R1843,R1844,R1845,R1846,R1847,R1848,R1849,R1849,R1850,R1851,R1852,R1853,R1854,R1855,R1856,R1857,R1858,R1859,R1859,R1860,R1861,R1862,R1863,R1864,R1865,R1866,R1867,R1868,R1869,R1869,R1870,R1871,R1872,R1873,R1874,R1875,R1876,R1877,R1878,R1879,R1879,R1880,R1881,R1882,R1883,R1884,R1885,R1886,R1887,R1888,R1889,R1889,R1890,R1891,R1892,R1893,R1894,R1895,R1896,R1897,R1898,R1898,R1899,R1899,R1900,R1899,R1901,R1902,R1903,R1904,R1905,R1906,R1907,R1908,R1909,R1909,R1910,R1911,R1912,R1913,R1914,R1915,R1916,R1917,R1918,R1919,R1919,R1920,R1921,R1922,R1923,R1924,R1925,R1926,R1927,R1928,R1929,R1929,R1930,R1931,R1932,R1933,R1934,R1935,R1936,R1937,R1938,R1939,R1939,R1940,R1941,R1942,R1943,R1944,R1945,R1946,R1947,R1948,R1949,R1949,R1950,R1951,R1952,R1953,R1954,R1955,R1956,R1957,R1958,R1959,R1959,R1960,R1961,R1962,R1963,R1964,R1965,R1966,R1967,R1968,R1969,R1969,R1970,R1971,R1972,R1973,R1974,R1975,R1976,R1977,R1978,R1979,R1979,R1980,R1981,R1982,R1983,R1984,R1985,R1986,R1987,R1988,R1989,R1989,R1990,R1991,R1992,R1993,R1994,R1995,R1996,R1997,R1998,R1998,R1999,R1999,R2000,R1999,R2001,R2002,R2003,R2004,R2005,R2006,R2007,R2008,R2009,R2009,R2010,R2011,R2012,R2013,R2014,R2015,R2016,R2017,R2018,R2019,R2019,R2020,R2021,R2022,R2023,R2024,R2025,R2026,R2027,R2028,R2029,R2029,R2030,R2031,R2032,R2033,R2034,R2035,R2036,R2037,R2038,R2039,R2039,R2040,R2041,R2042,R2043,R2044,R2045,R2046,R2047,R2048,R2049,R2049,R2050,R2051,R2052,R2053,R2054,R2055,R2056,R2057,R2058,R2059,R2059,R2060,R2061,R2062,R2063,R2064,R2065,R2066,R2067,R2068,R2069,R2069,R2070,R2071,R2072,R2073,R2074,R2075,R2076,R2077,R2078,R2079,R2079,R2080,R2081,R2082,R2083,R2084,R2085,R2086,R2087,R2088,R2089,R2089,R2090,R2091,R2092,R2093,R2094,R2095,R2096,R2097,R2098,R2098,R2099,R2099,R2100,R2099,R2101,R2102,R2103,R2104,R2105,R2106,R2107,R2108,R2109,R2109,R2110,R2111,R2112,R2113,R2114,R2115,R2116,R2117,R2118,R2119,R2119,R2120,R2121,R2122,R2123,R2124,R2125,R2126,R2127,R2128,R2129,R2129,R2130,R2131,R2132,R2133,R2134,R2135,R2136,R2137,R2138,R2139,R2139,R2140,R2141,R2142,R2143,R2144,R2145,R2146,R2147,R2148,R2149,R2149

# Today

## ■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

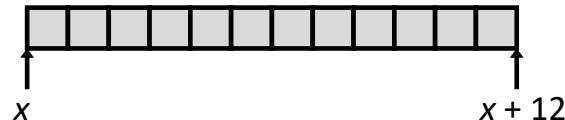
# Array Allocation

## ■ Basic Principle

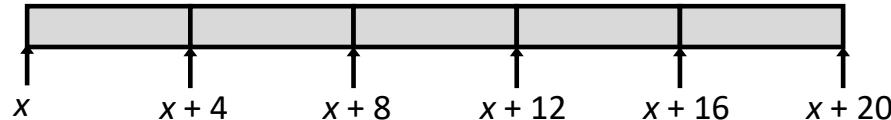
$T A[L];$

- Array of data type  $T$  and length  $L$
- Contiguously allocated region of  $L * \text{sizeof}(T)$  bytes in memory

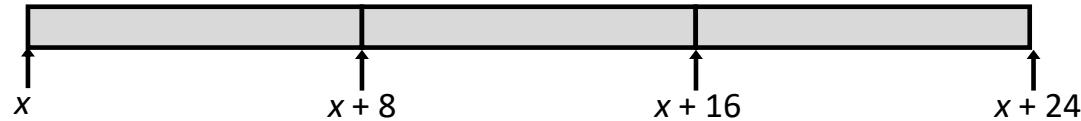
`char string[12];`



`int val[5];`



`double a[3];`



`char *p[3];`

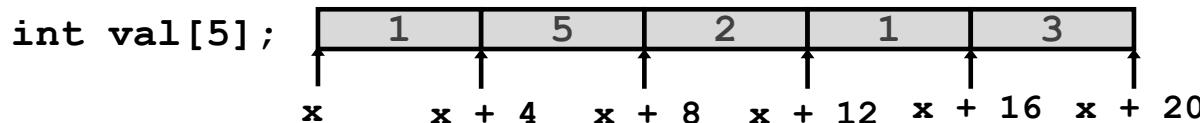


# Array Access

## ■ Basic Principle

$T \ A[L];$

- Array of data type  $T$  and length  $L$
- Identifier **A** can be used as a pointer to array element 0: Type  $T^*$



## ■ Reference      Type      Value

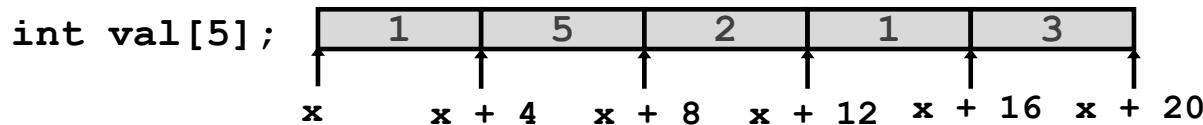
|                   |                |                   |                                                   |
|-------------------|----------------|-------------------|---------------------------------------------------|
| $\text{val}[4]$   | $\text{int}$   | 3                 | $\text{Val}[4] = \text{Val}^*[4] = \text{val}[4]$ |
| $\text{val}$      | $\text{int}^*$ | $\text{Val}$      | $\text{Val}$                                      |
| $\text{val}+1$    | $\text{int}^*$ | $\&\text{Val}[1]$ | $\&\text{Val}[1]$                                 |
| $\&\text{val}[2]$ | $\text{int}^*$ | $\text{Val}[2]$   | $\text{Val}[2]$                                   |
| $\text{val}[5]$   | $\text{int}$   | !                 | !                                                 |
| $*(\text{val}+1)$ | $\text{int}$   | $\text{Val}[1]$   | $\text{Val}[1]$                                   |
| $\text{val} + i$  | $\text{int}^*$ |                   |                                                   |

# Array Access

## ■ Basic Principle

$T \ A[L];$

- Array of data type  $T$  and length  $L$
- Identifier **A** can be used as a pointer to array element 0: Type  $T^*$



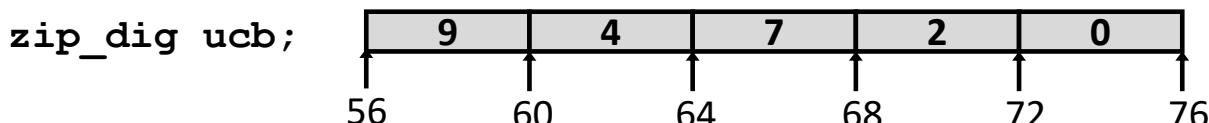
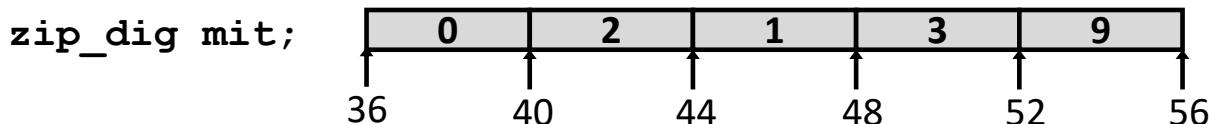
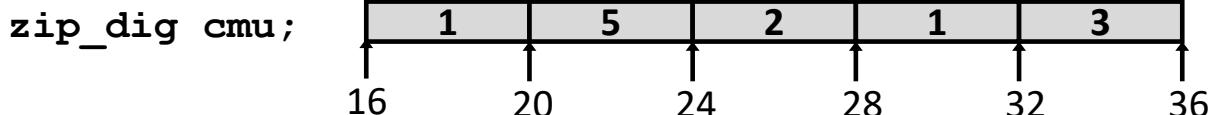
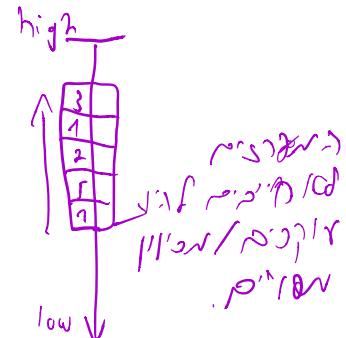
## ■ Reference      Type      Value

|                          |                    |                                      |
|--------------------------|--------------------|--------------------------------------|
| <code>val[4]</code>      | <code>int</code>   | 3                                    |
| <code>val</code>         | <code>int *</code> | <code>x</code>                       |
| <code>val+1</code>       | <code>int *</code> | <code>x + 4</code>                   |
| <code>&amp;val[2]</code> | <code>int *</code> | <code>x + 8</code>                   |
| <code>val[5]</code>      | <code>int</code>   | <code>??</code>                      |
| <code>* (val+1)</code>   | <code>int</code>   | 5 // <code>val[1]</code>             |
| <code>val + i</code>     | <code>int *</code> | <code>x + 4 * i //&amp;val[i]</code> |

# Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];
```

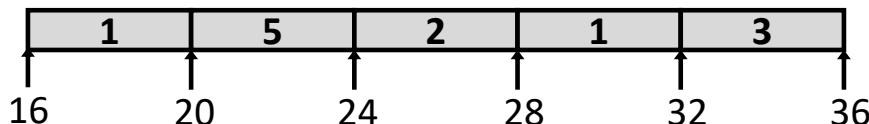
```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

# Array Accessing Example

```
zip_dig cmu;
```



```
int get_digit
    (zip_dig z, int digit)
{
    return z[digit];
}
```

## x86-64

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at  $\%rdi + 4 * \%rsi$
- Use memory reference `(%rdi,%rsi,4)`

# Array Loop Example

```
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

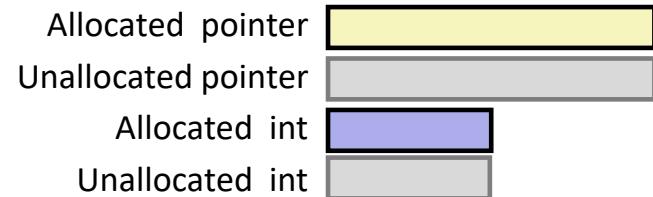
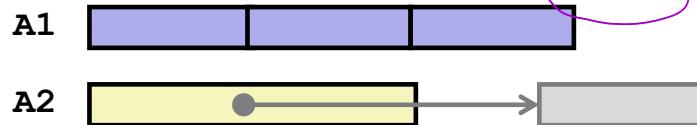
```
# %rdi = z
movl    $0, %eax
jmp     .L3
.L4:
    addl    $1, (%rdi,%rax,4)
    addq    $1, %rax
.L3:
    cmpq    $4, %rax
    jbe     .L4
    rep; ret
```

Annotations:

- A red arrow points from the `addl` instruction to the `$1, (%rdi,%rax,4)` operand.
- A purple arrow points from the `rep; ret` instruction to the `(%rdi,%rax,4)` operand.
- The text `i = 4` is written next to the purple arrow.

# Understanding Pointers & Arrays #1

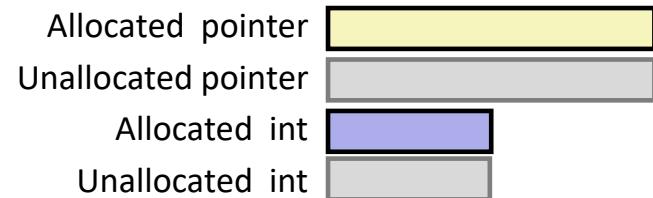
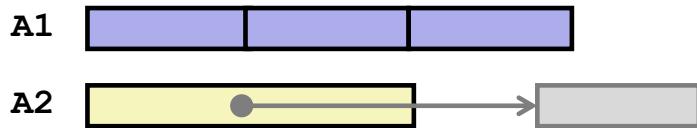
| Decl      | A1 , A2 |     |                        | *A1 , *A2 |     |      |
|-----------|---------|-----|------------------------|-----------|-----|------|
|           | Comp    | Bad | Size                   | Comp      | Bad | Size |
| int A1[3] | Y       | N   | 12 (3*4)<br>(S.O. int) | Y         | N   | 4    |
| int *A2   | Y       | N   | 8                      | Y         | Y   | 4    |



- Comp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

# Understanding Pointers & Arrays #1

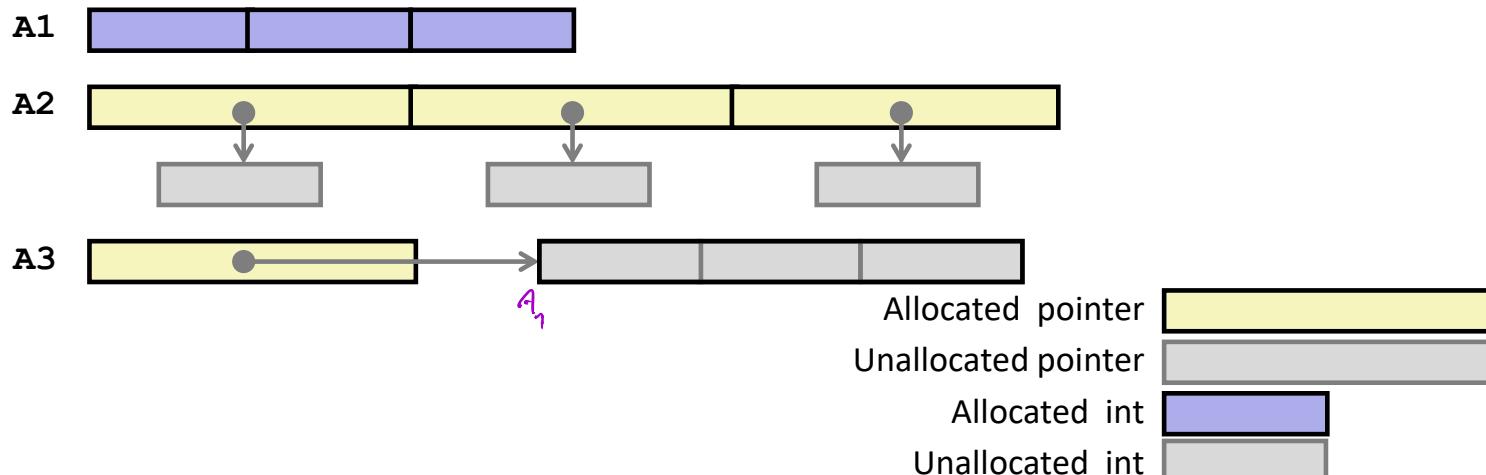
| Decl      | A1 , A2 |     |      | *A1 , *A2 |     |      |
|-----------|---------|-----|------|-----------|-----|------|
|           | Comp    | Bad | Size | Comp      | Bad | Size |
| int A1[3] | Y       | N   | 12   | Y         | N   | 4    |
| int *A2   | Y       | N   | 8    | Y         | Y   | 4    |



- **Comp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

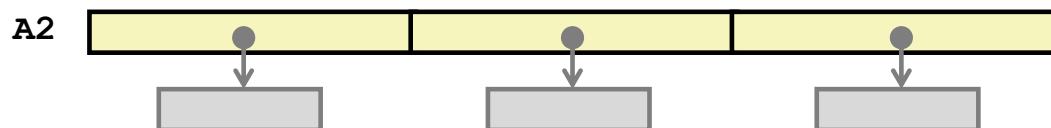
# Understanding Pointers & Arrays #2

| Decl                      | <i>An</i> |     |      | <i>*An</i> |     |      | <i>**An</i> |     |      |
|---------------------------|-----------|-----|------|------------|-----|------|-------------|-----|------|
|                           | Cmp       | Bad | Size | Cmp        | Bad | Size | Cmp         | Bad | Size |
| <code>int A1[3]</code>    | Y         | N   | 12   | X          | N   | 4    | N           | W   | W    |
| <code>int *A2[3]</code>   | Y         | N   | 24   | Y          | N   | 8??  | Y           | Y   | 4    |
| <code>int (*A3)[3]</code> | Y         | N   | 8    | Y          | Y   | 12   | Y           | Y   | 4    |



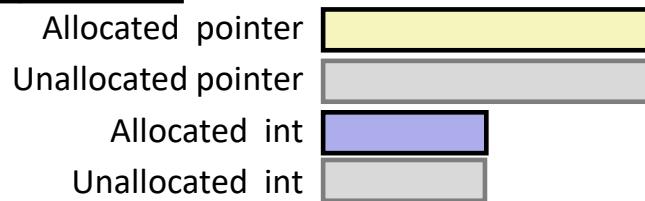
# Understanding Pointers & Arrays #2

| Decl                      | $A_n$ |     |      | $*A_n$ |     |      | $**A_n$ |     |      |
|---------------------------|-------|-----|------|--------|-----|------|---------|-----|------|
|                           | Cmp   | Bad | Size | Cmp    | Bad | Size | Cmp     | Bad | Size |
| <code>int A1[3]</code>    | Y     | N   | 12   | Y      | N   | 4    | N       | -   | -    |
| <code>int *A2[3]</code>   | Y     | N   | 24   | Y      | N   | 8 ?? | Y       | Y   | 4    |
| <code>int (*A3)[3]</code> | Y     | N   | 8    | Y      | Y   | 12   | Y       | Y   | 4    |



$A_3 = A_1$

$A_3 \leftarrow ??$



# Multidimensional (Nested) Arrays

## ■ Declaration

$T \ A[R][C];$

- 2D array of data type  $T$
- $R$  rows,  $C$  columns

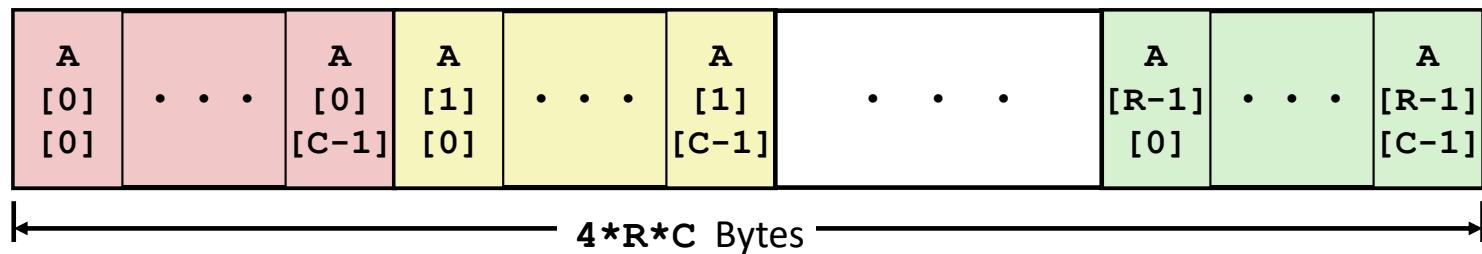
## ■ Array Size

- $R * C * \text{sizeof}(T)$  bytes

## ■ Arrangement

- Row-Major Ordering

`int A[R][C];`



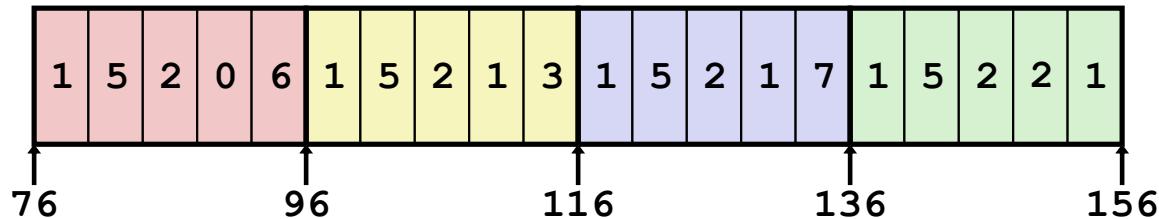
# Nested Array Example

```
#define PCOUNT 4
typedef int zip_dig[5];

zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

(use Row-major rest)  
1 3 2 8 5 6 3  
2 4 5 7 1 6 2

zip\_dig  
pgh[4];



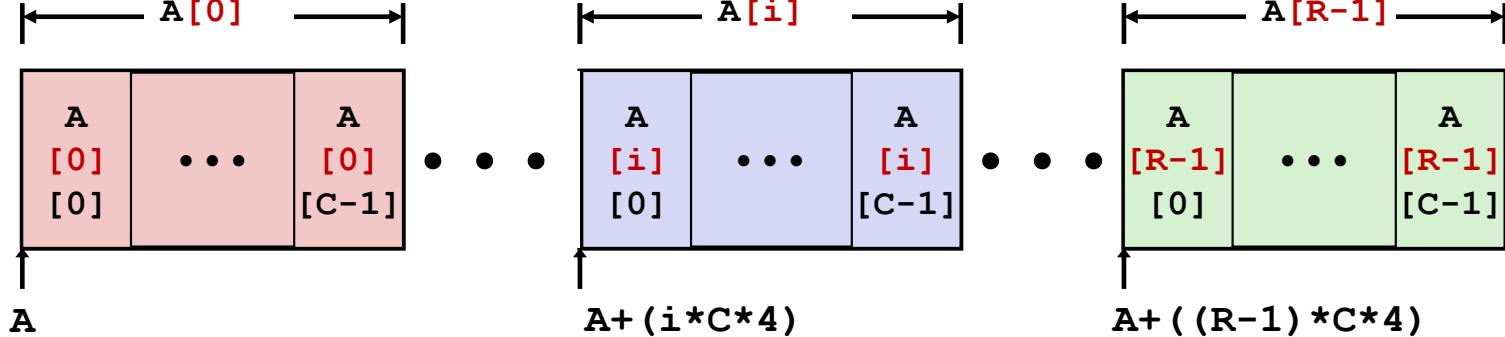
- “`zip_dig pgh[4]`” equivalent to “`int pgh[4][5]`”
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`’s, allocated contiguously
- “**Row-Major**” ordering of all elements in memory

# Nested Array Row Access

## ■ Row Vectors

- $\mathbf{A[i]}$  is array of  $C$  elements of type  $T$
- Starting address  $\mathbf{A}_{(0)} + i * (C * \text{sizeof}(T))$

```
int A[R][C];
```

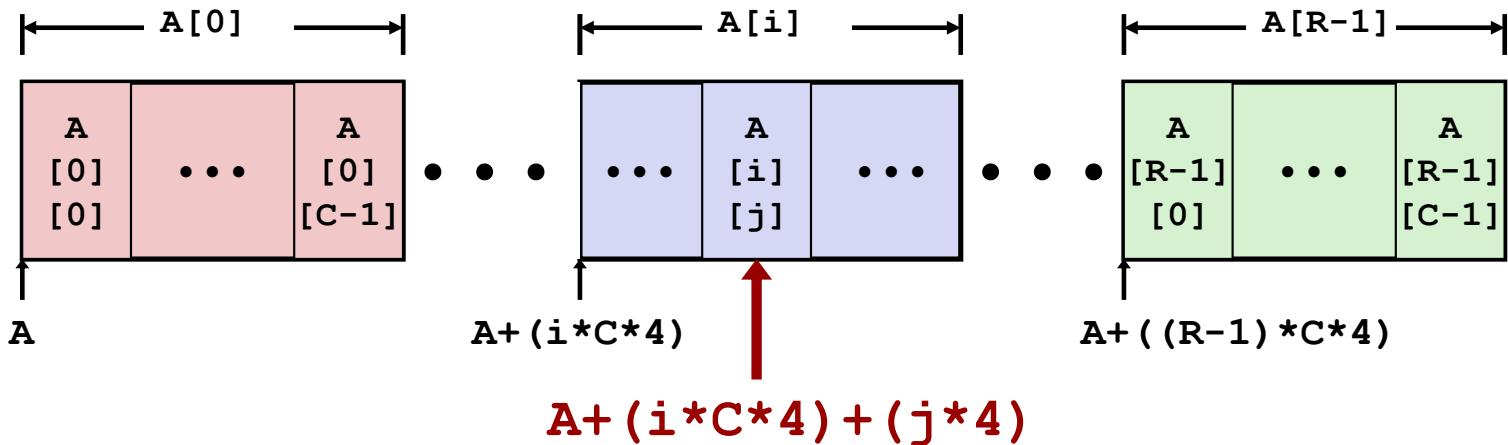


# Nested Array Element Access

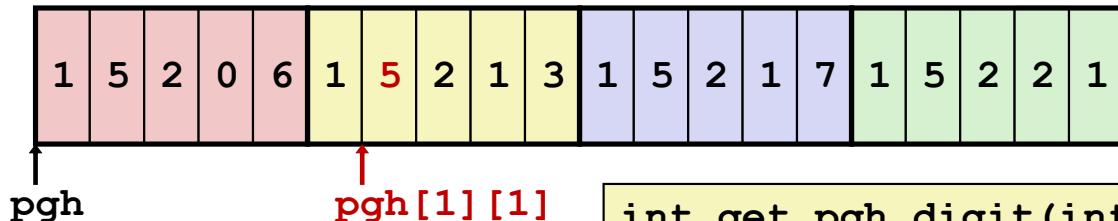
## ■ Array Elements

- $A[i][j]$  is element of type  $T$ , which requires  $K$  bytes
  - Address  $A + i * (C * K) + j * K$
- ( $i$  is row index)  
 (row  $i$ )  
 ( $j$  is column index)  
 $= A + (i * C + j) * K$

```
int A[R][C];
```



# Nested Array Element Access Code



```
int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```
leaq (%rdi,%rdi,4), %rax    # 5*index
addl %rax, %rsi              # 5*index+dig
movl pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

Sizeof int

## ■ Array Elements

- `pgh[index][dig]` is `int`
- Address:  $pgh + 20*index + 4*dig$   
 $= pgh + 4*(5*index + dig)$

# Array indexing (yes/no)

1. Access to a 2d array requires knowing the column length  $N$  ✓
2. ~~int~~ Arr [N] [M] is equivalent to ~~typedef int T[N]; T~~ Arr [M]; X  $\cancel{N}$
3. Given a 3d array: int A[L] [M] [N];  $\underbrace{\text{Arr}["\text{arr}\text{[M]}"]}_{\text{Arr}[M]}$   
 $A[i][j][k]=A+(i*M*N+j*N+k)*sizeof(int);$  Y ✓
4. To allocate a local variable int A[10] [10] the following code is needed

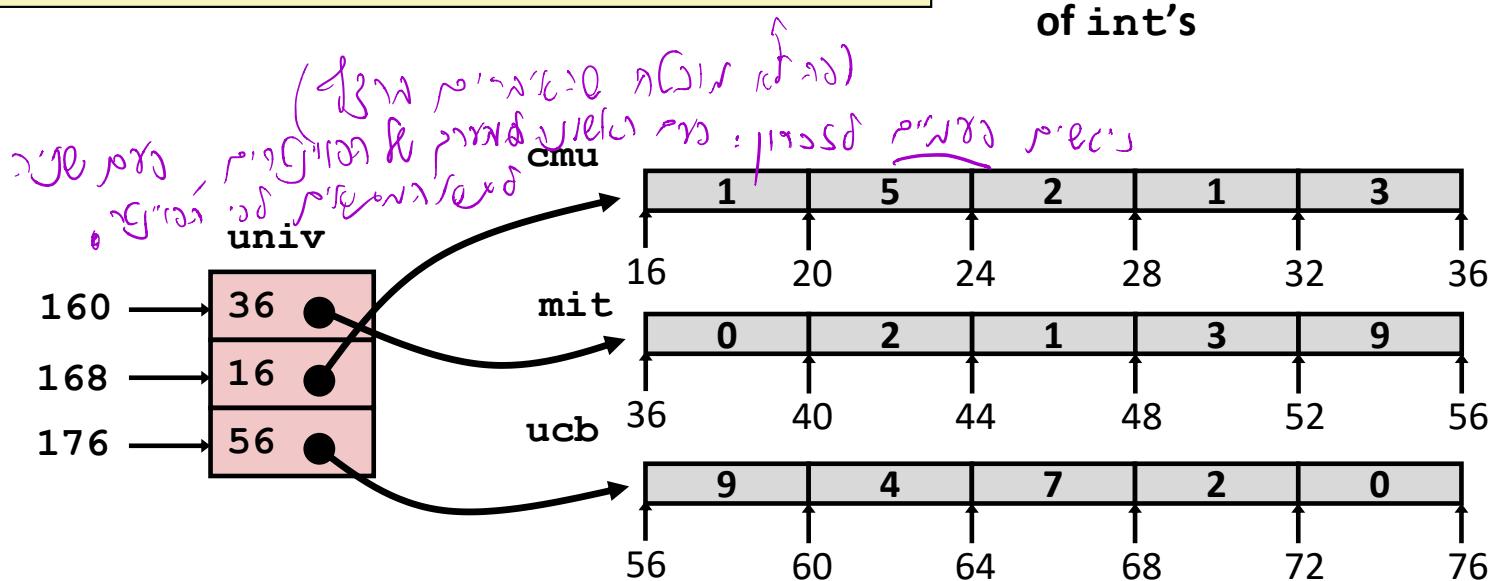
subq 10\*10\*4,%rsp Y  $\leftarrow$  [Initial 400 Stack + 10\*10\*4]

# Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 } ;
zip_dig mit = { 0, 2, 1, 3, 9 } ;
zip_dig ucb = { 9, 4, 7, 2, 0 } ;
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb} ;
```

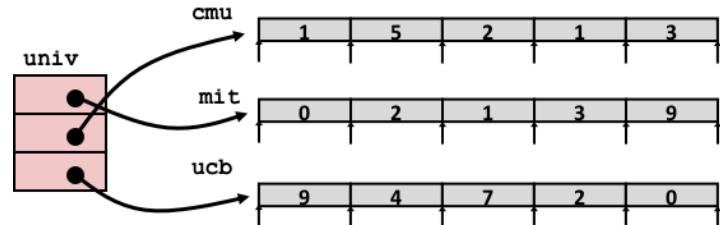
- Variable **univ** denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of **int's**



# Element Access in Multi-Level Array

```
int get_univ_digit
    (size_t index, size_t digit)
{
    return univ[index][digit];
}
```

(ANSWER ALREADY)



```
salq    $2, %rsi          # 4*digit
addq    univ(%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax       # return *p
ret
```

(ANSWER ALREADY)

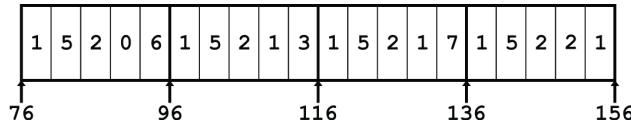
## Computation

- Element access **Mem[Mem[univ+8\*index]+4\*digit]**
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

# Array Element Accesses

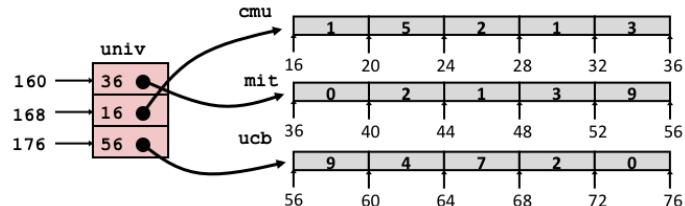
Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
    return pgh[index][digit];
}
```



Multi-level array

```
int get_univ_digit
  (size_t index, size_t digit)
{
    return univ[index][digit];
}
```



Accesses look similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`

`Mem[Mem[univ+8*index]+4*digit]`

# $N \times N$ Matrix

## Code

### ■ Fixed dimensions

- Know value of  $N$  at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
            size_t i, size_t j)
{
    return A[i][j];
}
```

### ■ Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
            size_t i, size_t j)
{
    return A[IDX(n,i,j)];
}
```

### ■ Variable dimensions, implicit indexing

- Now supported by gcc

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n],
            size_t i, size_t j) {
    return A[i][j];
}
```

# 16 X 16 Matrix Access

## ■ Array Elements

- `int A[16][16];`
- Address `A + i * (C * K) + j * K`
- $C = 16, K = 4$

```
/* Get element A[i][j] */
int fix_ele(fix_matrix A, size_t i, size_t j) {
    return A[i][j];
}
```

```
# A in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi           # 64*i
addq    %rsi, %rdi          # A + 64*i
movl    (%rdi,%rdx,4), %eax # Mem[A + 64*i + 4*j]
ret
```

# $n \times n$ Matrix Access

## ■ Array Elements

- `size_t n;`
- `int A[n][n];`
- Address  $A + i * (C * K) + j * K$
- $C = n, K = 4$
- Must perform integer multiplication

```
/* Get element A[i][j] */  
int var_ele(size_t n, int A[n][n], size_t i, size_t j)  
{  
    return A[i][j];  
}
```

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx  
imulq    %rdx, %rdi          # n*i  
leaq     (%rsi,%rdi,4), %rax # A + 4*n*i  
movl     (%rax,%rcx,4), %eax # A + 4*n*i + 4*j  
ret
```

# Multi-level array

- Given:

```
typedef int Arr[3];
Arr a={1,2,3}; Arr b={4,5,6};
int *full[2]={a,b};
```

We claim that  $a[4]==b[2]==5?$  *N* ✓

- Number of memory accesses to access an element of a nested array is the same as the number of memory accesses to access an element in a 2D array *(C<sub>1,2</sub>)*

# Today

## ■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

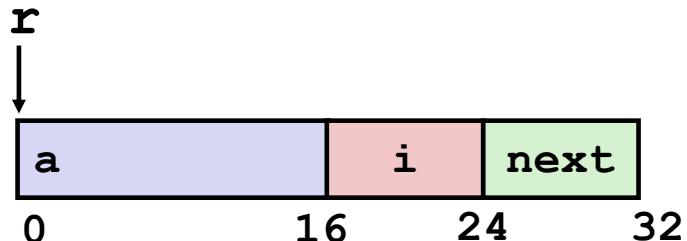
## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

# Structure Representation

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```

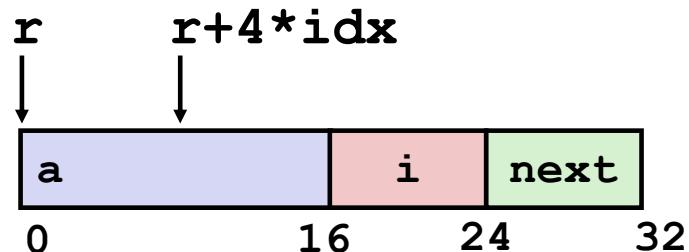


(pure union រួចរាល់ខ្លួនឱ្យ)

- Structure represented as block of memory
    - Big enough to hold all of the fields
  - Fields ordered according to declaration (ក្នុងក្រឡូវបានដោយណា)
- Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

# Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



## ■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as  $r + 4*idx$

```
int *get_ap
(struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```

```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

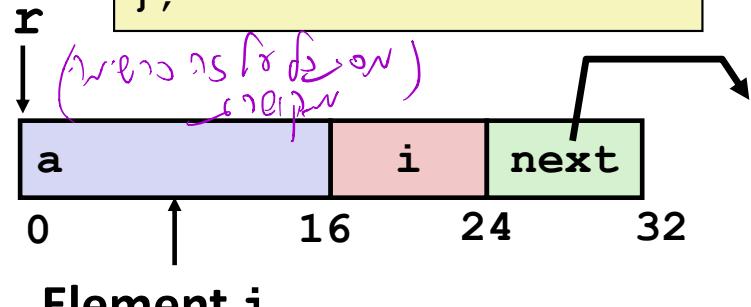
↑  
(STRUCT-> a[0])

# Following Linked List

## ■ C Code

```
void set_val
  (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



Element i

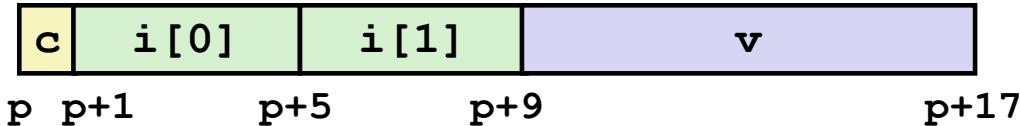
| Register | Value            |
|----------|------------------|
| %rdi     | <code>r</code>   |
| %rsi     | <code>val</code> |

```
.L11:                                # loop:
    movslq 16(%rdi), %rax      #   i = Mem[r+16]
    movl    %esi, (%rdi,%rax,4) #   Mem[r+4*i] = val
    movq    24(%rdi), %rdi      #   r = Mem[r+24]
    testq   %rdi, %rdi         #   Test r
    jne     .L11                #   if !=0 goto loop
```

# Structures & Alignment

"align" *fn*

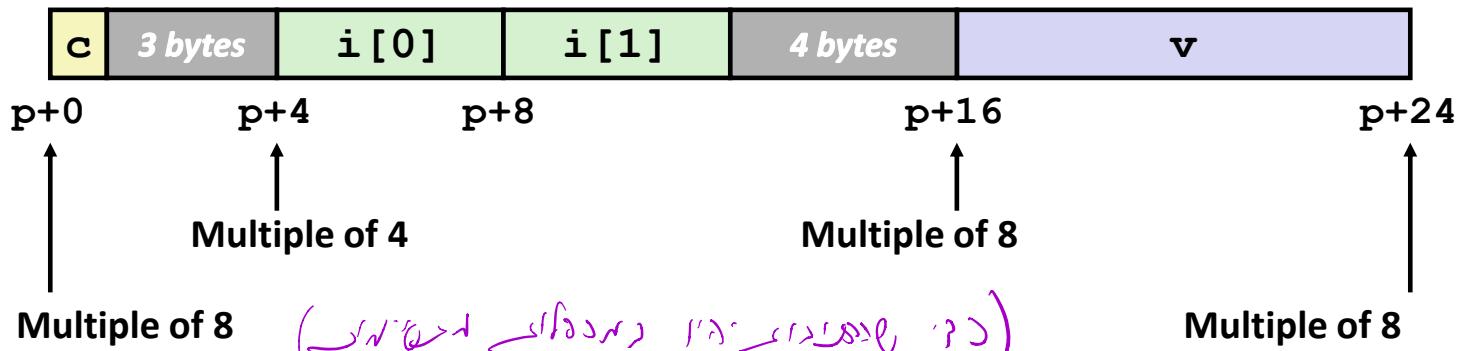
## ■ Unaligned Data



```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

## ■ Aligned Data

- Primitive data type requires  $B$  bytes implies  
Address must be multiple of  $B$



# Alignment Principles

پرینسپیل آف آلینجمنٹ

(سنچار میں اس فریم وکارڈ کو نہیں)

(اگرچہ اس کو فریم  
پریم جو بھی ہے تو  
کوئی پروپریتی نہیں)

## ■ Aligned Data

- Primitive data type requires  $B$  bytes
- Address must be multiple of  $B$
- Required on some machines; advised on x86-64

## ■ Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans cache lines (64 bytes). Intel states should avoid crossing 16 byte boundaries.

*[Cache lines will be discussed in Lecture 11.]*

- Virtual memory trickier when datum spans 2 pages (4 KB pages)

*[Virtual memory pages will be discussed in Lecture 17.]*

## ■ Compiler

- Inserts gaps in structure to ensure correct alignment of fields

# Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address
- **2 bytes: short, ...**
  - lowest 1 bit of address must be  $0_2$
- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be  $00_2$
- **8 bytes: double, long, char \*, ...**
  - lowest 3 bits of address must be  $000_2$

# Satisfying Alignment with Structures

## ■ Within structure:

- Must satisfy each element's alignment requirement

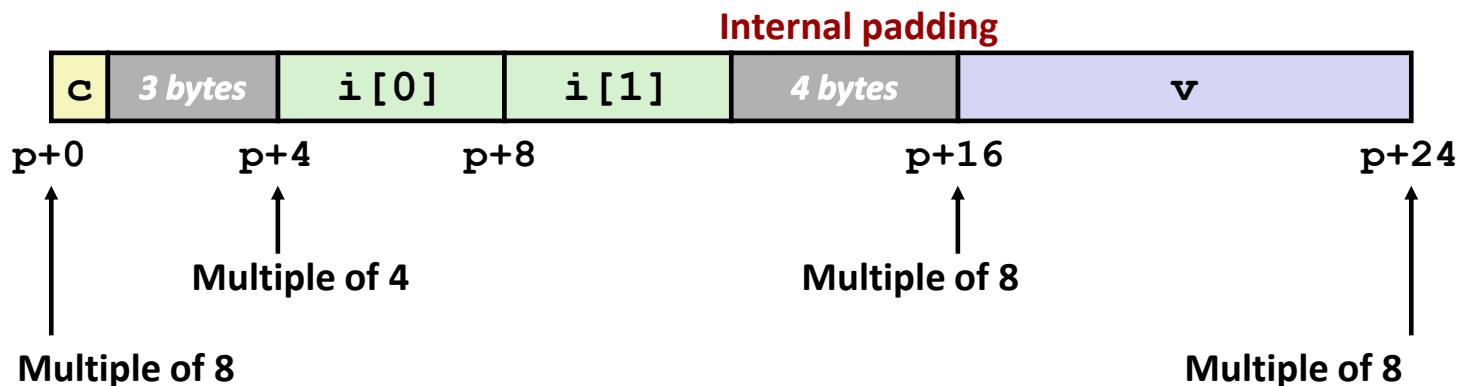
## ■ Overall structure placement

- Each structure has alignment requirement **K**
  - **K** = Largest alignment of any element
- Initial address & structure length must be multiples of **K**

## ■ Example:

- **K** = 8, due to **double** element

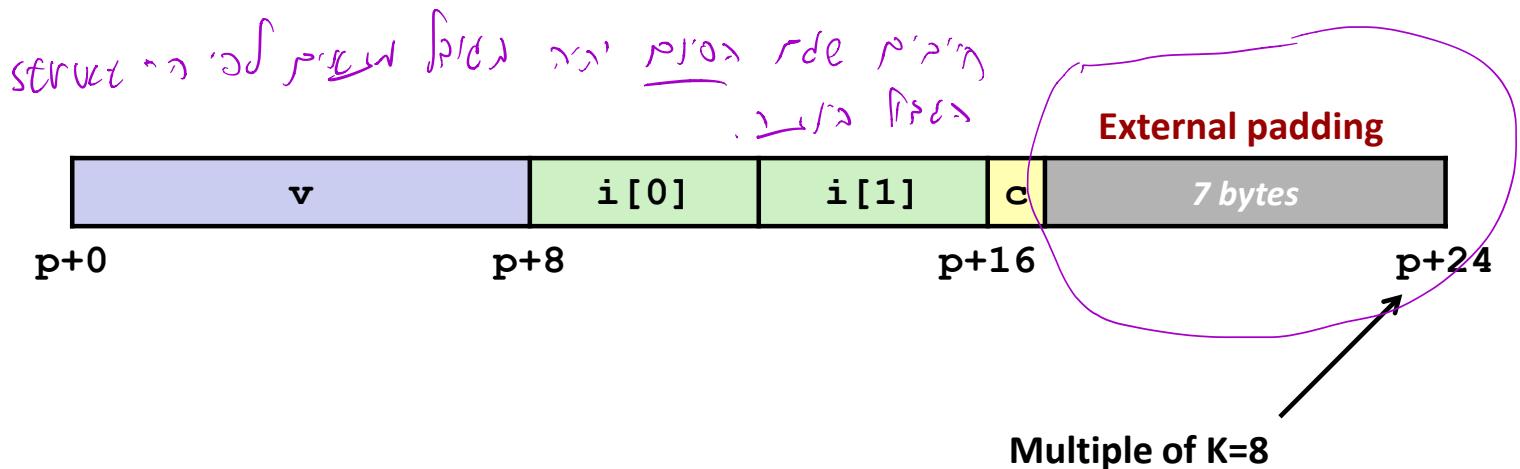
```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



# Meeting Overall Alignment Requirement

- For largest alignment requirement K
  - Overall structure must be multiple of K

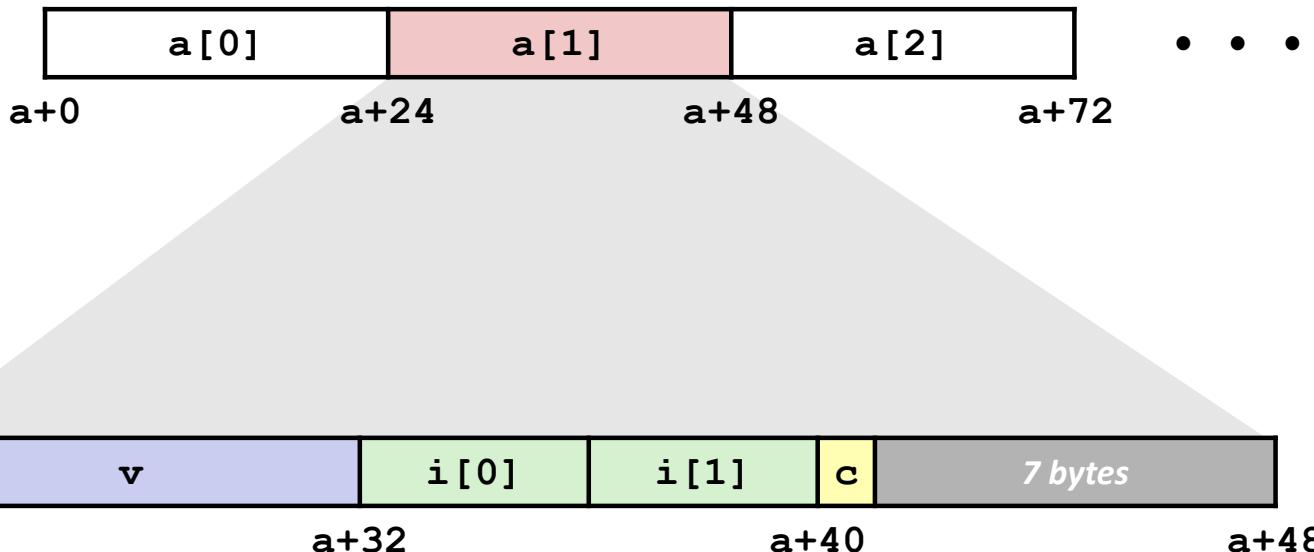
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



# Arrays of Structures

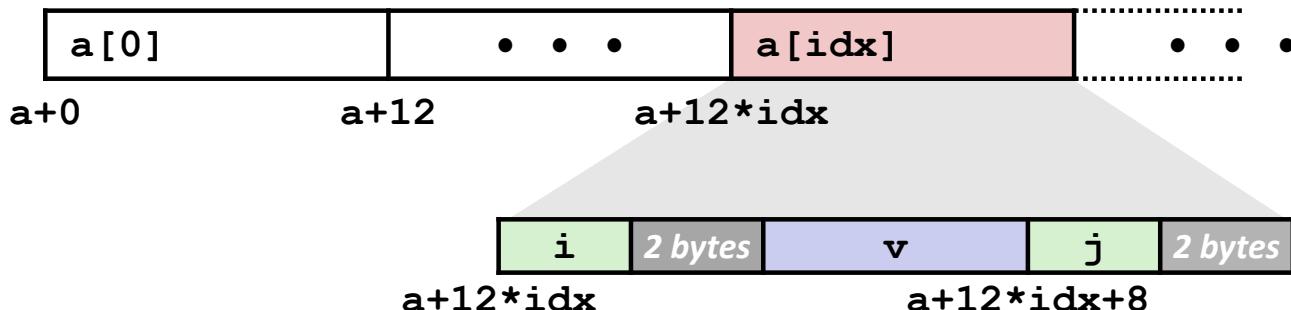
- Overall structure length  
multiple of K ( $\stackrel{= 24}{\text{15 bytes}}$ )
- Satisfy alignment requirement  
for every element

```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



# Accessing Array Elements

- Compute array offset  $12 * \text{idx}$ 
  - `sizeof(S3)`, including alignment spacers
- Element  $j$  is at offset 8 within structure
- Assembler gives offset  $a+8$ 
  - Resolved during linking



```
short get_j(int idx)
{
    return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(%rax,4),%eax
```

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

# Saving Space

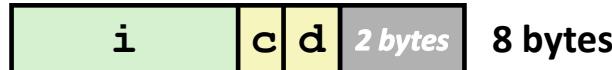
- Put large data types first

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```

```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

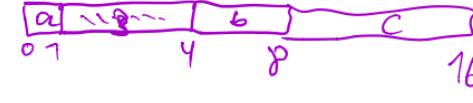


- Effect (largest alignment requirement K=4)



# Structs and alignment

```
struct S{ char a; int b; float c;} *s;
```

1.  $\&(s \rightarrow c) == s + 6$  ✓   $\Rightarrow$  
2.  $\&(s \rightarrow a) == s$  ✓
3.  $\&s == 0 \bmod 8$  ✓
4.  $\text{sizeof}(S) == 12$  ✓
5.  $S \text{ arr\_of\_s}[10]; \&(s[2].a) == s + \text{sizeof}(S) * 2$  ✓

Q.D.M:

• (JK 10/11/18) typedef γՅԱՆԿ՝ ԵՐԱ ՈՎՈՒՅԻՇՎԱՐ ԴՐԱՀԱՄ ՏԸՆԴՀԱՆՈՒՐ ՏԱՐԺ (1  
• (ՋԱԿԱՆ 5 ԲՆ ս-ին ԲՆ ՀԱՅԱՍՏԱՆԻ

Stacked Inception block (2)

לעתה נזכיר את היחסים בין המושגים הנ"ל:

(-0.15 1.00 -0.5) und ist int- $\delta$   $A_n \rightarrow 183.8102$  und für  $\delta$  (3)

מִצְרָיִם וְנַעֲמָן (nestp'd) (4)

$A[i][j] \dots [k]$ : קרא מעתה נסמן  $i$  ו-  $j$  ו-  $k$ . וילא פיזיקלי

$\Rightarrow A = \{i : j \dots k \in j \dots k\} \circ \text{pop}(\emptyset) / \{i\}$

• פְּסָנְדָה בְּרִיאָה, יְפֵנָה וְסַפְתָּא : פְּסָנְדָה (ק)

טבלה: היחסים בין מינרלים ורמות מינרלים נראים במ'קן (ז)

מתקן life, רעלס מוגניף ג'אנטס פלטפורם : multi level array (5)

١٠-١٢) (ب) حساب متحركة لـ  $\int_{\Gamma} \mathbf{F} \cdot d\mathbf{l}$ .

- 7' s — 13 p x f  $\rightarrow$  x k f  $\rightarrow$  p i p w  $\rightarrow$  ④ n : f'  $\rightarrow$  1 (16)

38)  $\lim_{x \rightarrow 0} \frac{\sin x}{x}$

• الكلمة المركبة هي الكلمة التي تتكون من مقطعين أو أكثر، وهي تكتب ككلمة واحدة.

We know from structures for acids  $\text{H}_3\text{O}^+$

מ'ג STRUCTS פונקציית סטראktור, שפירושה מילויים

جیاں کوئی ملکہ نہیں اور جو کوئی ملکہ نہیں تو وہ کوئی ملکہ نہیں۔

הנישׁוּתָה בְּבִירַעַם וְבְבֵיתָם כְּפָרְסָה וְבְבֵיתָם כְּפָרְסָה

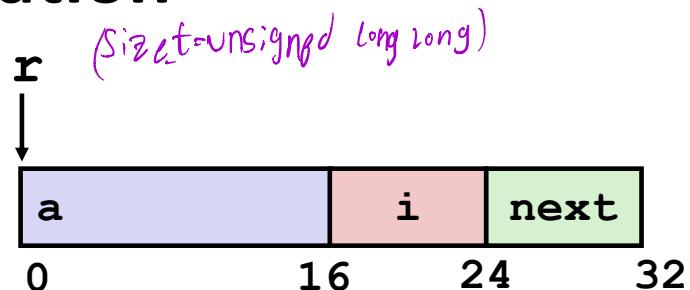
(ج) ملک مختاری کا درجہ ۱۰۰ فٹ میں مکانیکی وسائل کا انتظام

הארה מילא נספחים לאלה שנקראים כוונתית  
( נספחים )  
7) כוונתית נספחים נספחים נספחים נספחים  
... ( str. - } str. str. str. str. str. str. str. str.

# Structure Representation

*(writing)*

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

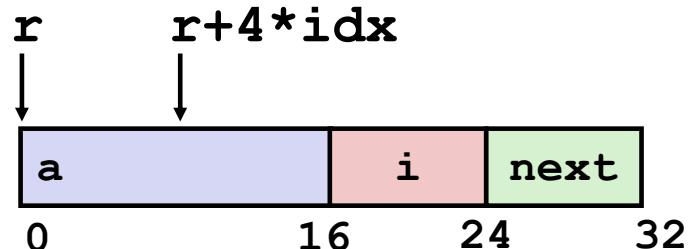
Struct R;

R.a[1] (year) is a "j0  
a p8m")

: If r &c  
r->i == (\*R).i

# Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



## ■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as  $r + 4*idx$

```
int *get_ap
(struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```

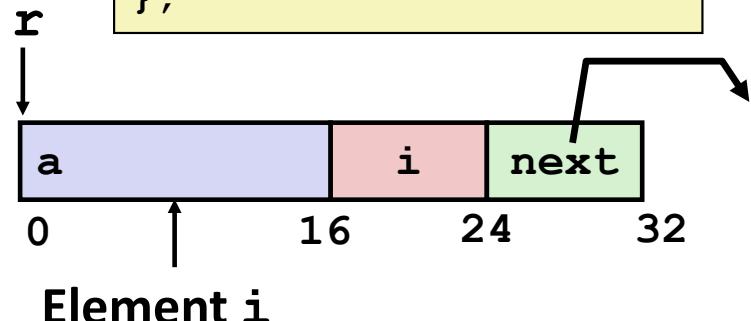
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

# Following Linked List

## ■ C Code

```
void set_val
  (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



Element i

| Register | Value |
|----------|-------|
| %rdi     | r     |
| %rsi     | val   |

```
.L11:                                # loop:
    movslq 16(%rdi), %rax      #   i = Mem[r+16]
    movl    %esi, (%rdi,%rax,4) #   Mem[r+4*i] = val
    movq    24(%rdi), %rdi      #   r = Mem[r+24]
    testq   %rdi, %rdi         #   Test r
    jne     .L11                #   if !=0 goto loop
```

# Structs

```
struct S{ double d; char c[2]; } *s ;
```

```
s=malloc(sizeof(struct S));
```

1. &(s->d) == s ? *y* ✓
2. &(\*s).d == s->d ? *n* ✓
3. if s == 0x1000 then s++ == 0x1001 ? *n* ✓

# Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
- **Unions**

# x86-64 Linux Memory Layout

## ■ Stack

- Runtime stack (8MB limit)
- E. g., local variables

## ■ Heap

- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new()`

## ■ Data

- Statically allocated data
- E.g., global vars, `static` vars, string constants

## ■ Text / Shared Libraries

- Executable machine instructions
- Read-only

*Upwards*

00007FFFFFFFFF

00007FFF00000000

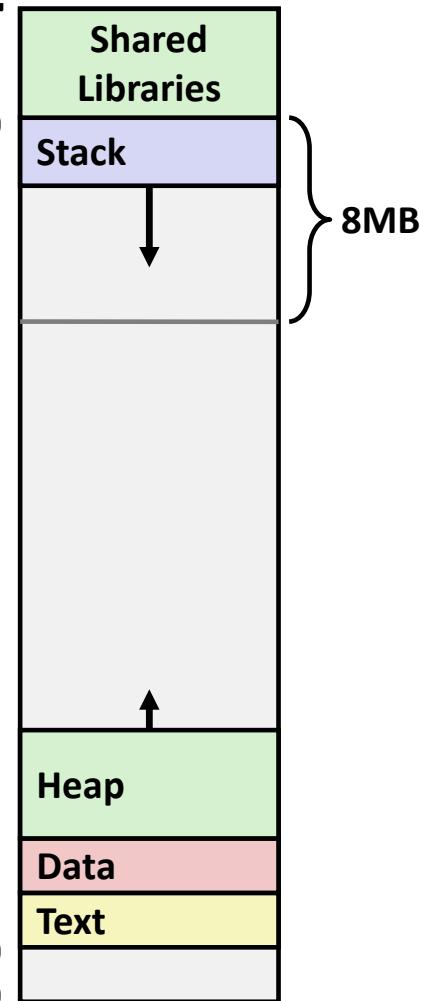
*Downwards*

Hex Address

400000  
000000



*not drawn to scale*



*not drawn to scale*

# Memory Allocation Example

00007FFFFFFFFF

```

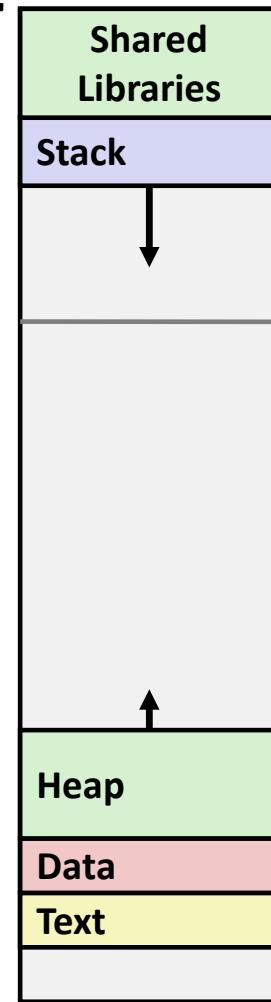
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main () (Stack ↑)
{ (Heap → [Re]Alloc [P]tr)
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}

```



*Where does everything go?*

# x86-64 Example Addresses

*address range  $\sim 2^{47}$*

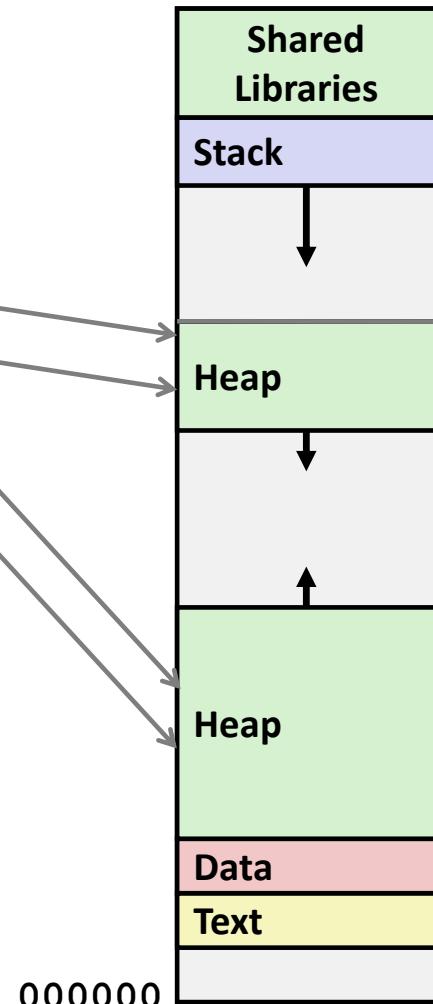
|            |                    |
|------------|--------------------|
| local      | 0x00007ffe4d3be87c |
| p1         | 0x00007f7262a1e010 |
| p3         | 0x00007f7162a1d010 |
| p4         | 0x000000008359d120 |
| p2         | 0x000000008359d010 |
| big_array  | 0x0000000080601060 |
| huge_array | 0x0000000000601060 |
| main()     | 0x000000000040060c |
| useless()  | 0x0000000000400590 |

*data {*

|                    |
|--------------------|
| 0x00007ffe4d3be87c |
| 0x00007f7262a1e010 |
| 0x00007f7162a1d010 |
| 0x000000008359d120 |
| 0x000000008359d010 |
| 0x0000000080601060 |
| 0x0000000000601060 |
| 0x000000000040060c |
| 0x0000000000400590 |

*possible seg faults -> heap -> >> no plan / s  
 (return address pt -> user j/c)*

*not drawn to scale*

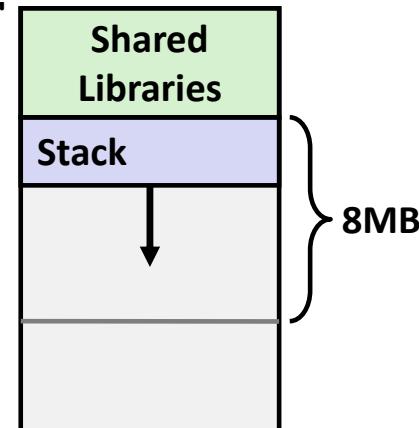


not drawn to scale

# Runaway Stack Example

00007FFFFFFFFF

```
int recurse(int x) {
    int a[1<<15]; // 4*2^15 = 128 KiB
    printf("x = %d. a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```



- leads to stack overflow because it exceeds the limit of the stack*
- Functions store local data on in stack frame
- Recursive functions cause deep nesting of frames

```
./runaway 67
x = 67. a at 0x7ffd18aba930
x = 66. a at 0x7ffd18a9a920
x = 65. a at 0x7ffd18a7a910
x = 64. a at 0x7ffd18a5a900
. . .
x = 4. a at 0x7ffd182da540
x = 3. a at 0x7ffd182ba530
x = 2. a at 0x7ffd1829a520
Segmentation fault (core dumped)
```

# Runaway stack

צורך זריף נגזרות  
לפניהם יתאפשרו

1. Exceeding the stack space of a single function causes segmentation fault ✓ ✗
2. Issuing too many malloc calls may exhaust the stack ✓ (ריצועים רבים יתאפשרו)
3. Declaring large array as a local variable may cause segmentation fault ✓ ✓
4. Unbounded recursion is guaranteed to exhaust stack space ✓ ✓



# Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

# Recall: Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```



|                     |                    |                                |
|---------------------|--------------------|--------------------------------|
| <code>fun(0)</code> | <code>-&gt;</code> | <code>3.1400000000</code>      |
| <code>fun(1)</code> | <code>-&gt;</code> | <code>3.1400000000</code>      |
| <code>fun(2)</code> | <code>-&gt;</code> | <code>3.1399998665</code>      |
| <code>fun(3)</code> | <code>-&gt;</code> | <code>2.0000006104</code>      |
| <code>fun(6)</code> | <code>-&gt;</code> | <b>Stack smashing detected</b> |
| <code>fun(8)</code> | <code>-&gt;</code> | <b>Segmentation fault</b>      |

- Result is system specific

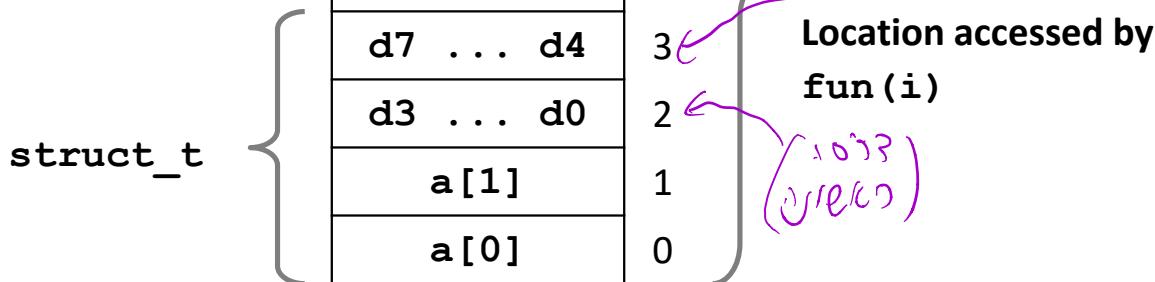
# Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;
```

|        |    |                    |
|--------|----|--------------------|
| fun(0) | -> | 3.1400000000       |
| fun(1) | -> | 3.1400000000       |
| fun(2) | -> | 3.1399998665       |
| fun(3) | -> | 2.0000006104       |
| fun(4) | -> | Segmentation fault |
| fun(8) | -> | 3.1400000000       |

## Explanation:

POSIX thread leak 'n  
overwriting R  
(overwriting D1 D2)



# Such problems are a **BIG** deal

- Generally called a “buffer overflow”
  - when exceeding the memory size allocated for an array
- Why a big deal?
  - It’s the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

# String Library Code

## ■ Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

ןַעֲשֵׂה אֶת הַסְּרִיר  
 מִלְּמָדָה קָדוֹשָׁה  
 שֶׁבָּאָתָה בְּפָנֵינוּ  
 וְאַתָּה תַּעֲשֵׂה  
 כַּאֲמָתָךְ כְּבָאָתָה  
 וְאַתָּה תַּעֲשֵׂה

- No way to specify limit on number of characters to read
- Similar problems with other library functions
  - `strcpy`, `strcat`: Copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

(0xd9999999)

E

←btw, how big  
is big enough?

```
void call_echo() {
    echo();
}
```

```
unix> ./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

```
unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Segmentation Fault
```

# Buffer Overflow Disassembly

echo:

```
00000000004006cf <echo>:
4006cf: 48 83 ec 18
4006d3: 48 89 e7
4006d6: e8 a5 ff ff ff
4006db: 48 89 e7
4006de: e8 3d fe ff ff
4006e3: 48 83 c4 18
4006e7: c3
```

(buf->128 stack for pw -> end)

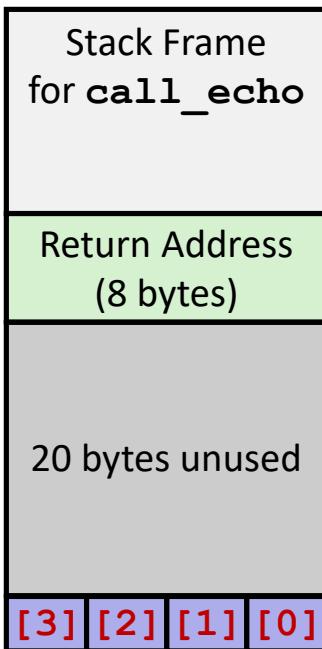
|                                                                                                                                |                                                |
|--------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| sub \$0x18,%rsp<br>mov %rsp,%rdi<br>callq 400680 <gets><br>mov %rsp,%rdi<br>callq 400520 <puts@plt><br>add \$0x18,%rsp<br>retq | (buf[32,24])<br>(buf[128,128])<br>(stack) [r1] |
|--------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|

call\_echo:

|                                                                                                                     |                                                                                   |
|---------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| 4006e8: 48 83 ec 08<br>4006ec: b8 00 00 00 00<br>4006f1: e8 d9 ff ff ff<br><b>4006f6:</b> 48 83 c4 08<br>4006fa: c3 | sub \$0x8,%rsp<br>mov \$0x0,%eax<br>callq 4006cf <echo><br>add \$0x8,%rsp<br>retq |
|---------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|

# Buffer Overflow Stack

*Before call to gets*

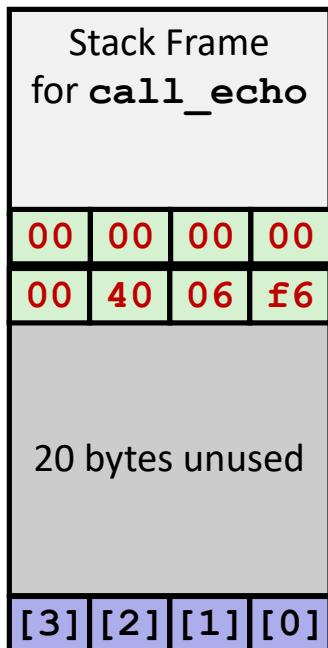


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    . . .
```

# Buffer Overflow Stack Example

*Before call to gets*



```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $x18, %rsp
    movq %rsp, %rdi
    call gets
    ...
}
```

`call_echo:`

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

# Buffer Overflow Stack Example #1

*After call to gets*

| Stack Frame<br>for <code>call_echo</code> |    |    |    |
|-------------------------------------------|----|----|----|
| 00                                        | 00 | 00 | 00 |
| 00                                        | 40 | 06 | f6 |
| 00                                        | 32 | 31 | 30 |
| 39                                        | 38 | 37 | 36 |
| 35                                        | 34 | 33 | 32 |
| 31                                        | 30 | 39 | 38 |
| 37                                        | 36 | 35 | 34 |
| 33                                        | 32 | 31 | 30 |

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    ...

```

`call_echo:`

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

`buf ← %rsp`

```
unix>./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

"01234567890123456789012\0"

**Overflowed buffer, but did not corrupt state**

17/257 3/9 6/10 15/12

# Buffer Overflow Stack Example #2

*After call to gets*

| Stack Frame<br>for <code>call_echo</code> |    |    |    |
|-------------------------------------------|----|----|----|
| 00                                        | 00 | 00 | 00 |
| 00                                        | 40 | 06 | 00 |
| 33                                        | 32 | 31 | 30 |
| 39                                        | 38 | 37 | 36 |
| 35                                        | 34 | 33 | 32 |
| 31                                        | 30 | 39 | 38 |
| 37                                        | 36 | 35 | 34 |
| 33                                        | 32 | 31 | 30 |

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
subq $24, %rsp
movq %rsp, %rdi
call gets
. . .
```

`call_echo:`

```
. .
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
. . .
```

`buf ← %rsp`

```
unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Segmentation fault
```

Program “returned” to 0x0400600, and then crashed.

# Stack Smashing Attacks

```
void P() {
    Q();
    ...
}
```

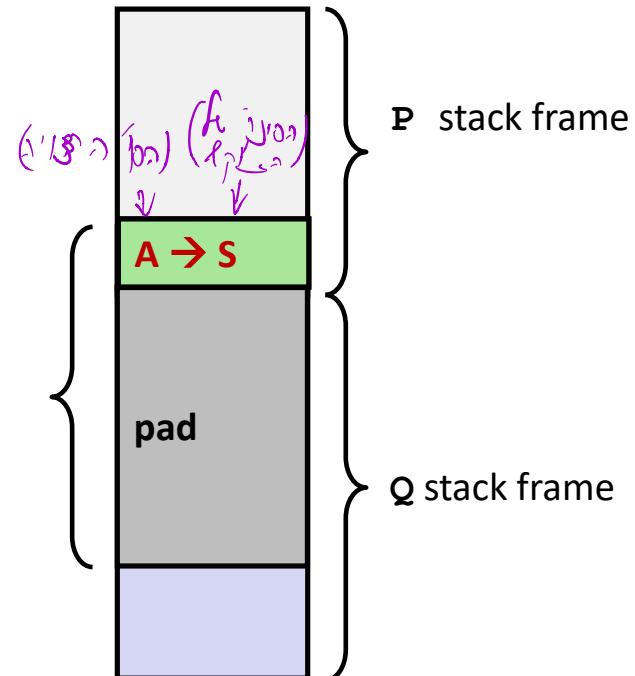
return address  
A

```
int Q() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```

```
void S() {
    /* Something
       unexpected */
    ...
}
```

data written by `gets()`

Stack after call to `gets()`



- Overwrite normal return address A with address of some other code S
- When Q executes `ret`, will jump to other code

# Crafting Smashing String

| Stack Frame<br>for <code>call echo</code> |    |    |    |
|-------------------------------------------|----|----|----|
| 00                                        | 00 | 00 | 00 |
| 00                                        | 48 | 83 | 80 |
| 00                                        | 00 | 00 | 00 |
| 00                                        | 40 | 06 | fb |
|                                           |    |    |    |
|                                           |    |    |    |
|                                           |    |    |    |
|                                           |    |    |    |
|                                           |    |    |    |
|                                           |    |    |    |

```
int echo() {
    char buf[4];
    gets(buf);
    ...
    return ...;
}
```

← %rsp

24 bytes

*Target Code*

```
void smash() {
    printf("I've been smashed!\n");
    exit(0);
}
```

00000000004006fb <smash>:  
4006fb: 48 83 ec 08

(F5D) o.?)

*Attack String (Hex)*

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 30 | 31 | 32 | 33 |
| fb | 06 | 40 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

little endian (SCEA)

# Smashing String Effect

| Stack Frame<br>for <code>call echo</code> |    |    |    |
|-------------------------------------------|----|----|----|
| 00                                        | 00 | 00 | 00 |
| 00                                        | 48 | 83 | 80 |
| 00                                        | 00 | 00 | 00 |
| 00                                        | 40 | 06 | fb |
| 33                                        | 32 | 31 | 30 |
| 39                                        | 38 | 37 | 36 |
| 35                                        | 34 | 33 | 32 |
| 31                                        | 30 | 39 | 38 |
| 37                                        | 36 | 35 | 34 |
| 33                                        | 32 | 31 | 30 |

← %rsp

*Target Code*

```
void smash() {  
    printf("I've been smashed!\n");  
    exit(0);  
}
```

00000000004006fb <smash>:  
4006fb: 48 83 ec 08

*Attack String (Hex)*

30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33  
fb 06 40 00 00 00 00 00

# Code Injection Attacks

לעוקב אחר פונקציית `gets()` ופונקציית `scanf()` בC מושגנו  
הנתקן מפונקציית `gets()` ופונקציית `scanf()` מפונקציית `main()`  
בC מושגנו מפונקציית `gets()` ופונקציית `scanf()` מפונקציית `main()`

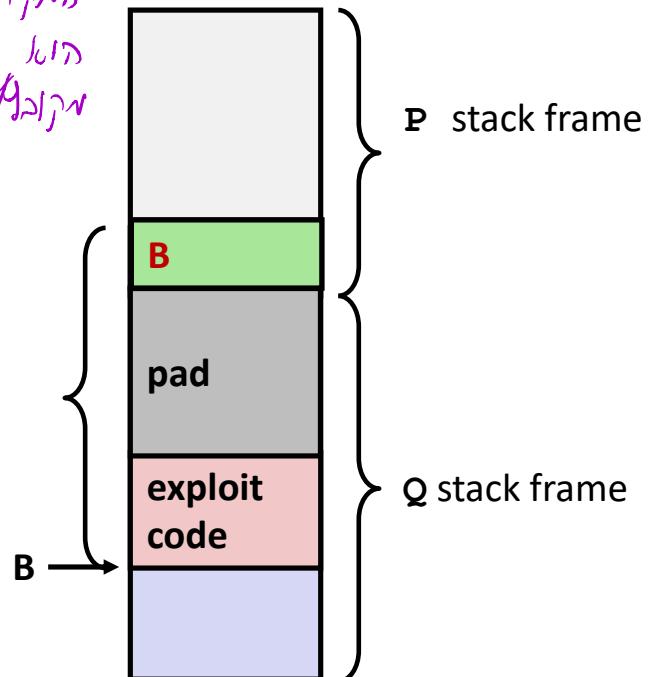
```
void P() {
    Q();
    ...
}
```

return  
address  
**A**

```
int Q() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```

data written  
by `gets()`

Stack after call to `gets()`



- Input string contains byte representation of executable code
- Overwrite return address `A` with address of buffer `B`
- When `Q` executes `ret`, will jump to exploit code

# How Does The Attack Code Execute?

```
void P() {  
    Q();  
    ...  
}
```

```
int Q() {  
    char buf[64];  
    gets(buf); // A->B  
    ...  
    return ...;  
}
```

ret

ret

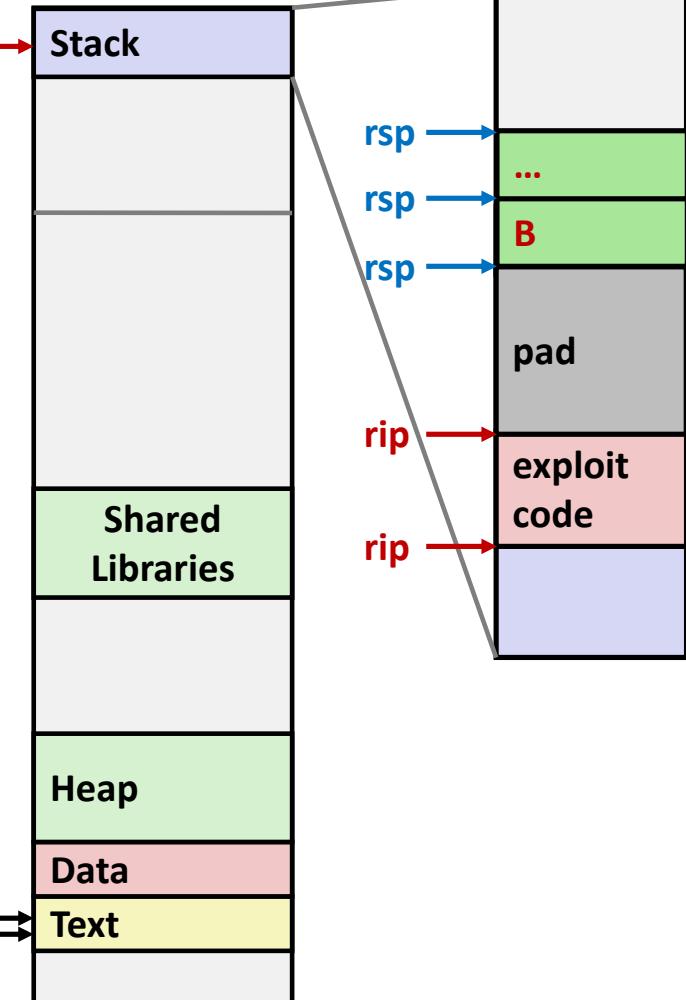
rip

rip

rip

rip

rip



# Buffer overflow

(4 GB)

void foo(){ int \*a=malloc(1L<<32); char c[5]; gets(c);}  
(چنانچه بزرگ دادن به a میتواند باعث اسکریپت هجوم شود)

1. The following input:"abcde" will create buffer overflow y ✓
2. For the buffer overflow attack to occur the input must be more than 4GB n ✓  
(heap خواهد مalloc کرد) چنانچه اگر a را بزرگ فرموده باشیم ۵ کاراکتر را در میز ۳۲ بایتی خواهی داشت
3. An attack overwrites the code that performs i++ n ✓ (برای این کار stack را هجوم کنید)
4. An attack modifies the return value of bar n ✓

و اگر این دادن جای a را در دستور کنترل کنیم، راکت -2 را میتوانیم تغییر داد  
و آنرا کنترل کنیم

# What To Do About Buffer Overflow Attacks

避免 Java C/C++

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each...

# 1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- For example, use library routines that limit string lengths
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use **%ns** where **n** is a suitable integer

## 2. System-Level Protections can help

### ■ Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Shifts stack addresses for entire program
- Makes it difficult for hacker to predict beginning of inserted code
- E.g.: 5 executions of memory allocation code

local

0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

- Stack repositioned each time program executes

(A) *Program State*  
 (B) *Exploit Code*

Stack base

Random allocation

main

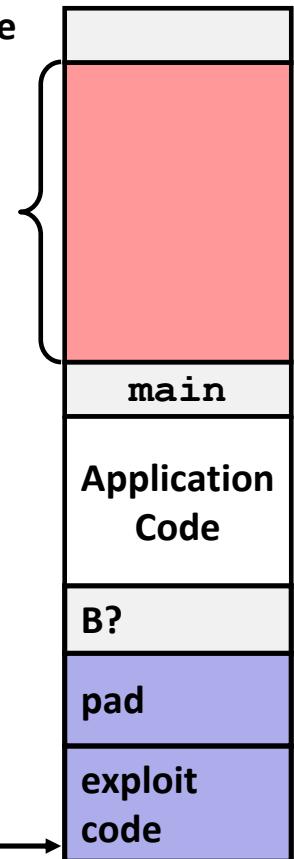
Application Code

B?

pad

exploit code

B?



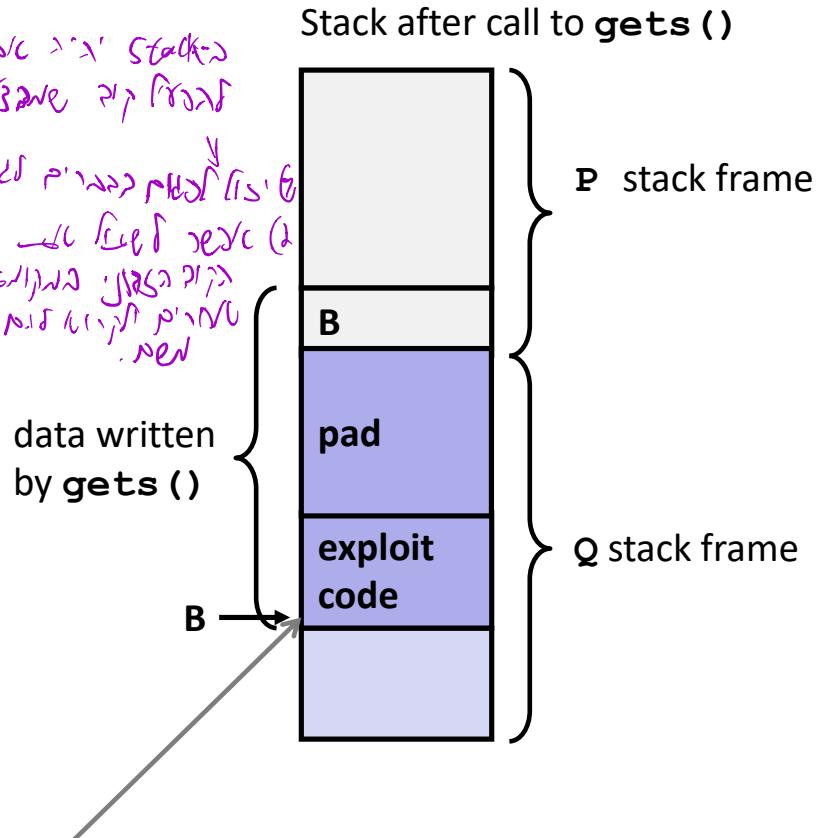
# 2. System-Level Protections can help

## ■ Nonexecutable code segments

- In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- x86-64 added explicit “execute” permission
- Stack marked as non-executable

Stack after call to `gets()`

Stack after `gets()`  
B  
pad  
exploit code



Any attempt to execute this code will fail

# 3. Stack Canaries can help

## ■ Idea

- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

## ■ GCC Implementation

- **-fstack-protector**
- Now the default (disabled earlier)

```
unix> ./bufdemo-sp
Type a string: 0123456
0123456
```

```
unix> ./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```

# Protected Buffer Disassembly

echo:

```

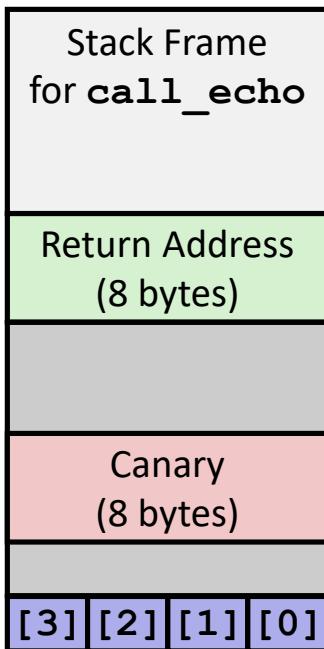
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: mov    0x8(%rsp),%rax
400758: xor    %fs:0x28,%rax
400761: je    400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq

```

(ptrs now) (ptrs)  
 (ptrs now) (ptrs)

# Setting Up Canary

*Before call to gets*

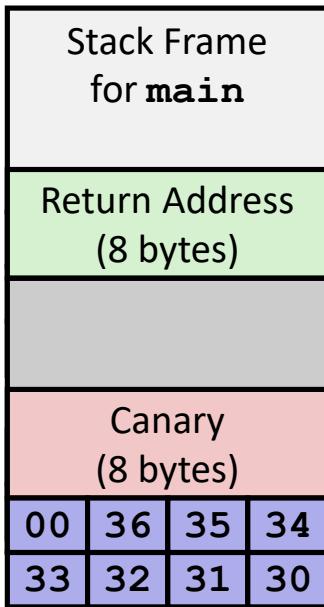


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
. . .
    movq    %fs:40, %rax    # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
. . .
```

# Checking Canary

*After call to gets*



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

**Input: 0123456**

```
echo:
. . .
movq    8(%rsp), %rax      # Retrieve from stack
xorq    %fs:40, %rax       # Compare to canary
je     .L6                  # If same, OK
call   __stack_chk_fail    # FAIL
```

# Buffer overflow protection

1. Stack canaries cannot prevent overflow, but only detect it ✓
2. Controlling the effective size of user input will eliminate buffer overflow problems ✓ (fixed max size, no risk to user)  
✓ (fixed max size, no risk to user)
3. Randomizing stack location helps because an attacker does not know where the malicious code is injected ✓
4. Marking stack as non-executable prevents only code-injection attacks but not buffer overflows in general ✓  
  
attacks

protects memory from being written to or executed by the program  
→ if the memory is not executable, the program can't execute the injected code  
→ (executing shellcode) → can't inject shellcode

! / \ 2 w d i f f i n g o w f o r

# Return-Oriented Programming Attacks

## ■ Challenge (for hackers)

- Stack randomization makes it hard to predict buffer location
- Marking stack nonexecutable makes it hard to insert binary code

## ■ Alternative Strategy

- Use existing code
  - E.g., library code from stdlib
- String together fragments to achieve overall desired outcome
- *Does not overcome stack canaries*

## ■ Construct program from *gadgets*

- Sequence of instructions ending in `ret`
  - Encoded by single byte `0xc3`
- Code positions fixed from run to run
- Code is executable

# Gadget Example #1

```
long ab_plus_c  
  (long a, long b, long c)  
{  
    return a*b + c;  
}
```

```
00000000004004d0 <ab_plus_c>:  
 4004d0: 48 0f af fe  imul %rsi,%rdi  
 4004d4: 48 8d 04 17  lea (%rdi,%rdx,1),%rax  
 4004d8: c3             retq
```

$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

# Gadget Example #2

```
void setval(unsigned *p) {  
    *p = 3347663060u;  
}
```

<setval>:

|                  |          |                           |
|------------------|----------|---------------------------|
| 4004d9: c7 07 d4 | 48 89 c7 | movl \$0xc78948d4, (%rdi) |
| 4004df: c3       |          | retq                      |

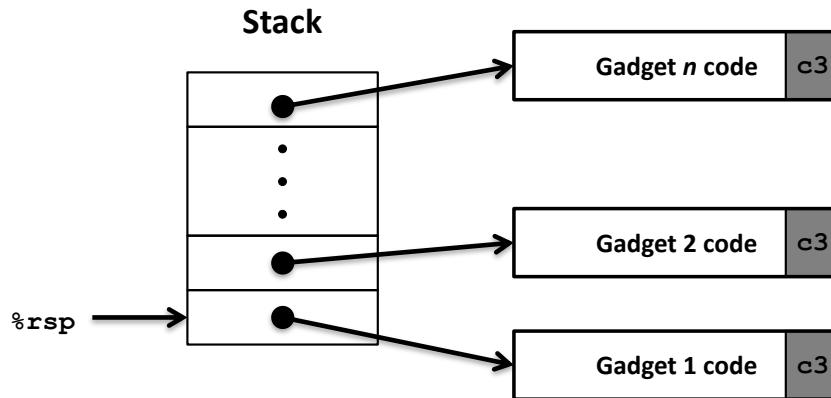
Encodes `movq %rax, %rdi`

rdi ← rax

Gadget address = 0x4004dc

## ■ Repurpose byte codes

# ROP Execution



- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one

# Crafting an ROB Attack String

| Stack Frame<br>for <code>call echo</code> |    |    |    |
|-------------------------------------------|----|----|----|
| 00                                        | 00 | 00 | 00 |
| 00                                        | 48 | 83 | 80 |
| 00                                        | 00 | 00 | 00 |
| 00                                        | 40 | 06 | f6 |
| 33                                        | 32 | 31 | 30 |
| 39                                        | 38 | 37 | 36 |
| 35                                        | 34 | 33 | 32 |
| 31                                        | 30 | 39 | 38 |
| 37                                        | 36 | 35 | 34 |
| 33                                        | 32 | 31 | 30 |

buf

## Gadget

```
00000000004004d0 <ab_plus_c>:
```

```
4004d0: 48 0f af fe imul %rsi,%rdi
```

```
4004d4: 48 8d 04 17 lea (%rdi,%rdx,1),%rax
```

```
4004d8: c3 retq
```

$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Attack: `int echo() returns rdi + rdx`

```
int echo() {
    char buf[4];
    gets(buf);
    ...
    return ...;
}
```

## Attack String (Hex)

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 30 | 31 | 32 | 33 |
| d4 | 04 | 40 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

Multiple gadgets will corrupt stack upwards

~~1015.0~~

→ Binaрна мембрсајт ојсј (битар) тај проприети је његова вредност structs (1)  
(a->key) → пратија, поинтер-ају је његова вредност (a.key) :)

$$a \rightarrow \text{key} := (*a). \text{key} \quad (=$$

: Segments -> final 11.25 (1) : memory layout (2)

- If you review the runtime-pass, 8MB stack (1c)

• (`(int malloc()`) ~~လုပ်မှု~~ ပြန်သော မျက်းများ ငါးမီး heap (၁)

PDF reader uses OCR tools : data (L)

• ۲۰۱۷-۲۰۱۸ میلادی هیئت‌کاری ایرانیان در گروه text / Shared libraries (۷)

heap  $\rightarrow$  ג'נְלָה גַּדְגַּדְתִּים malloc  $\rightarrow$  פֶּרֶס פֶּרֶס PC, טְמִינָה

( $\text{P16K} \cap \text{P3N}$ ) ejeciona pijón desde puj: Ruina Stack (3)

Stacked memory model uses multiple stacks for multiple threads.

• **Segmentation** für die Segmentation für die

Java fe Objekts fe  $10^{12}$  Krediten: Werk: Vulnerabilities (4)

• תרגום מילויים במשפטים קומתיים יפה

کوئی جائزی نہ کروں.

缓冲区溢出 - find for scanf into 'p10' : buffer overflow (5)

"EOF" - это конец файла, а "NULL" - конец строки.

الآن نحن في قلب المعركة

جایی کوئی نہیں بھاگ سکتا

- השלמה (1): השורה הראשונה (line 1) היא שורה שמייצגת stack. השורה השנייה (line 2) היא שורה שמייצגת canaries.
- השלמה (2): השורה הראשונה (line 1) היא שורה שמייצגת stack. השורה השנייה (line 2) היא שורה שמייצגת canaries.
- השלמה (3): השורה הראשונה (line 1) היא שורה שמייצגת stack. השורה השנייה (line 2) היא שורה שמייצגת canaries.
- השלמה (4): השורה הראשונה (line 1) היא שורה שמייצגת stack. השורה השנייה (line 2) היא שורה שמייצגת canaries.

# Today

- **Linking**
  - Motivation
  - What it does
  - How it works
  - Dynamic linking
- **Case study: Library interpositioning**

# Example C Program

(This function finds the sum of an array)

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

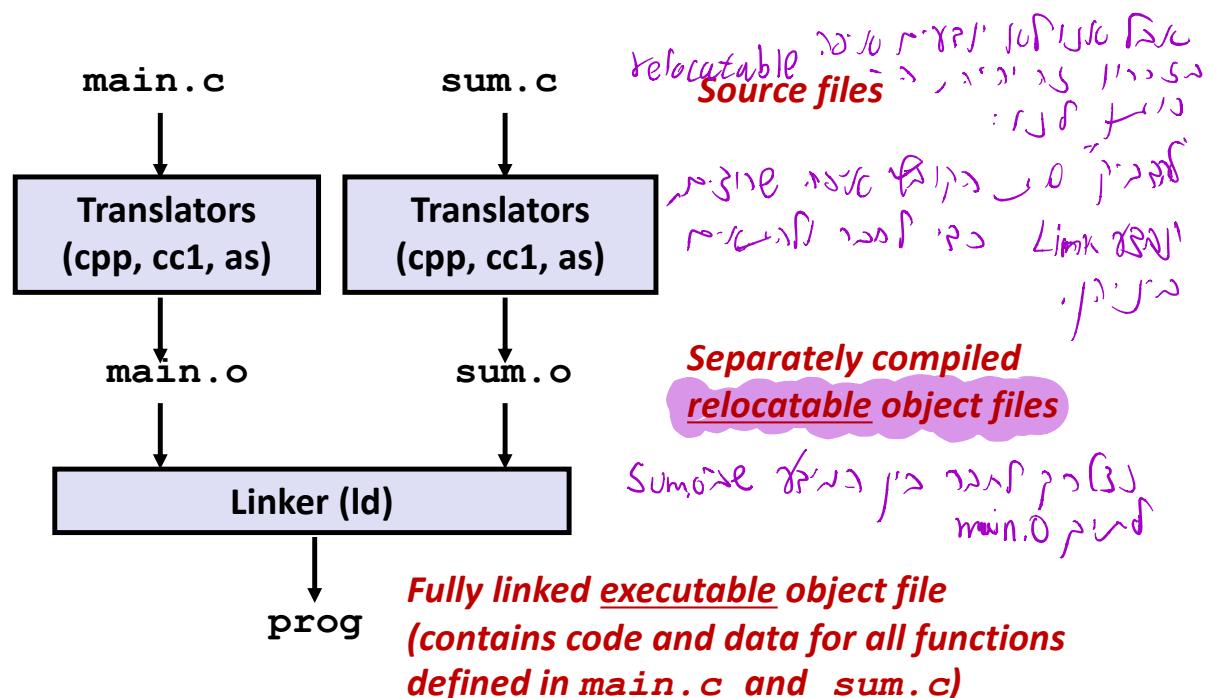
sum.c

Call  
(sum)  
→  
Temporary

For this program, we have:  
.data .text

# Linking

- Programs are translated and linked using a *compiler driver*:
  - linux> *gcc -Og -o prog main.c sum.c*
  - linux> *./prog*



# Why Linkers?

Programs today, multiple files linked for many purposes for single exec file

## ■ Reason 1: Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

(!) multiple files references to same symbols in different files

# Why Linkers? (cont)

## Reason 2: Efficiency

- Time: Separate compilation
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
  - Can compile multiple files concurrently.
- Space: Libraries
  - Common functions can be aggregated into a single file...
  - **Option 1: Static Linking** *הברך מילוי בזיכרון*
    - Executable files and running memory images contain only the library code they actually use
  - **Option 2: Dynamic linking** *הברך מילוי בזמן 실행*
    - Executable files contain no library code
    - During execution, single copy of library code can be shared across all executing processes

# Object files/separate compilation

1. If all the code is contained in one C file, the binary code will still be different in .o and the executable. *y ✓*
2. -Og flag is necessary to create object files *n ✓*
3. A.h is included in A.c, and B.h is included both in A.h and B.c. a user changed B.h. Only B.c should be recompiled *n ✓*
4. Linking is necessary to merge all .o into a single file in a certain format, but does not modify the instructions and their arguments *n ✓*

# What Do Linkers Do?

## ■ Step 1: Symbol resolution

- Programs define and reference symbols (global variables and functions):
  - `void swap() { ... } /* define symbol swap */`
  - `swap(); /* reference symbol swap */`
  - `int *xp = &x; /* define symbol xp, reference x */`
- Symbol definitions are stored in object file (by assembler) in *symbol table*.
  - Symbol table is an array of entries
  - Each entry includes name, size, and location of symbol.
- During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.

# Symbols in Example C Program

## Definitions

```
int sum(int *a, int n),  
  
int array[2] = {1, 2};  
  
int main(int argc, char** argv)  
{  
    int val = sum(array, 2);  
    return val;  
}
```

*main.c*

```
int sum(int *a, int n)  
{  
    int i, s = 0;  
  
    for (i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
}
```

*sum.c*

## Reference

# What Do Linkers Do? (cont)

## ■ Step 2: Relocation

- Merges separate code and data sections into single sections
- Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
- Updates all references to these symbols to reflect their new positions.

Let's look at these two steps in more detail....

# Three Kinds of Object Files (Modules)

(Հակոբյան Յան)

## ■ Relocatable object file (.o file)

- Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file

(Հակոբյան յան Արթուր Հայք)

## ■ Executable object file (a .out file)

- Contains code and data in a form that can be copied directly into memory and then executed.

## ■ Shared object file (.so file)

- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows

# Executable and Linkable Format (ELF)

(Windows の オブジェクト フォーマット は ELF だ)

- Standard binary format for object files
- One unified format for
  - Relocatable object files ( .o ),
  - Executable object files ( a.out )
  - Shared object files ( .so )
- Generic name: ELF binaries

# ELF Object File Format

אֶלְף אָבָטָה בְּרַקְעָן

## ■ Elf header

- Word size, byte ordering, file type (.o, exec, .so), machine type, etc.

## ■ Segment header table

- Page size, virtual addresses memory segments (sections), segment sizes.

## ■ .text section

- Code

## ■ .rodata section

- Read only data: jump tables, string constants, ...

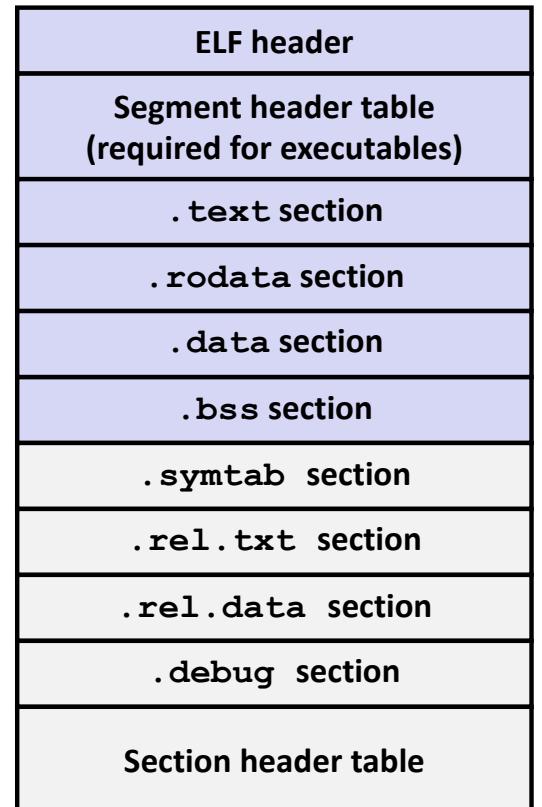
## ■ .data section

- Initialized global variables

## ■ .bss section

- Uninitialized global variables
- “Block Started by Symbol”
- “Better Save Space”

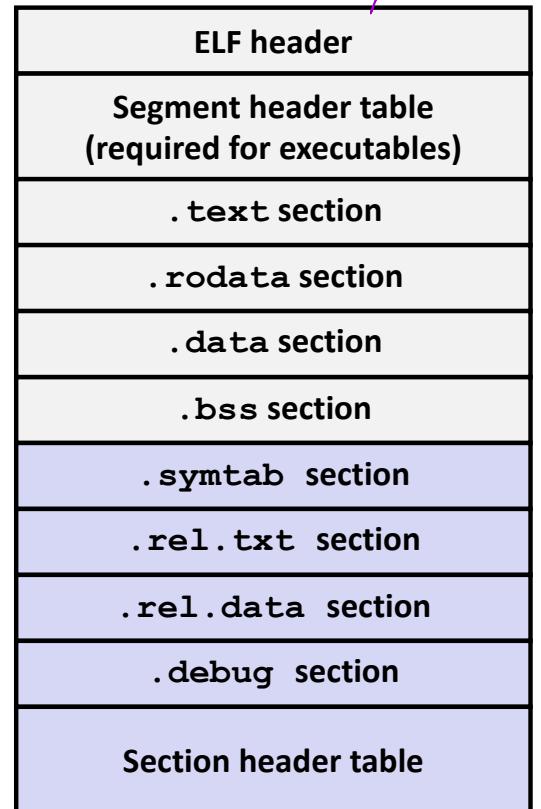
- Has section header but occupies no space



# ELF Object File Format (cont.)

- **.symtab section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **.rel.text section**
  - Relocation info for **.text** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **.rel.data section**
  - Relocation info for **.data** section
  - Addresses of pointer data that will need to be modified in the merged executable
- **.debug section**
  - Info for symbolic debugging (`gcc -g`)
- **Section header table**
  - Offsets and sizes of each section

(ר'פ'נ'א י'ס'ד Sourcecode ר'פ'נ'ה י'ס'ד ג'ז'ען ד'ג'ז'ה)  
-O ו'נ'ז'ב א'ז'ג ת'ב'ז'ה)



# Linker Symbols

## Global symbols

- Symbols defined by module  $m$  that can be referenced by other modules.
- E.g.: non-**static** C functions and non-**static** global variables.

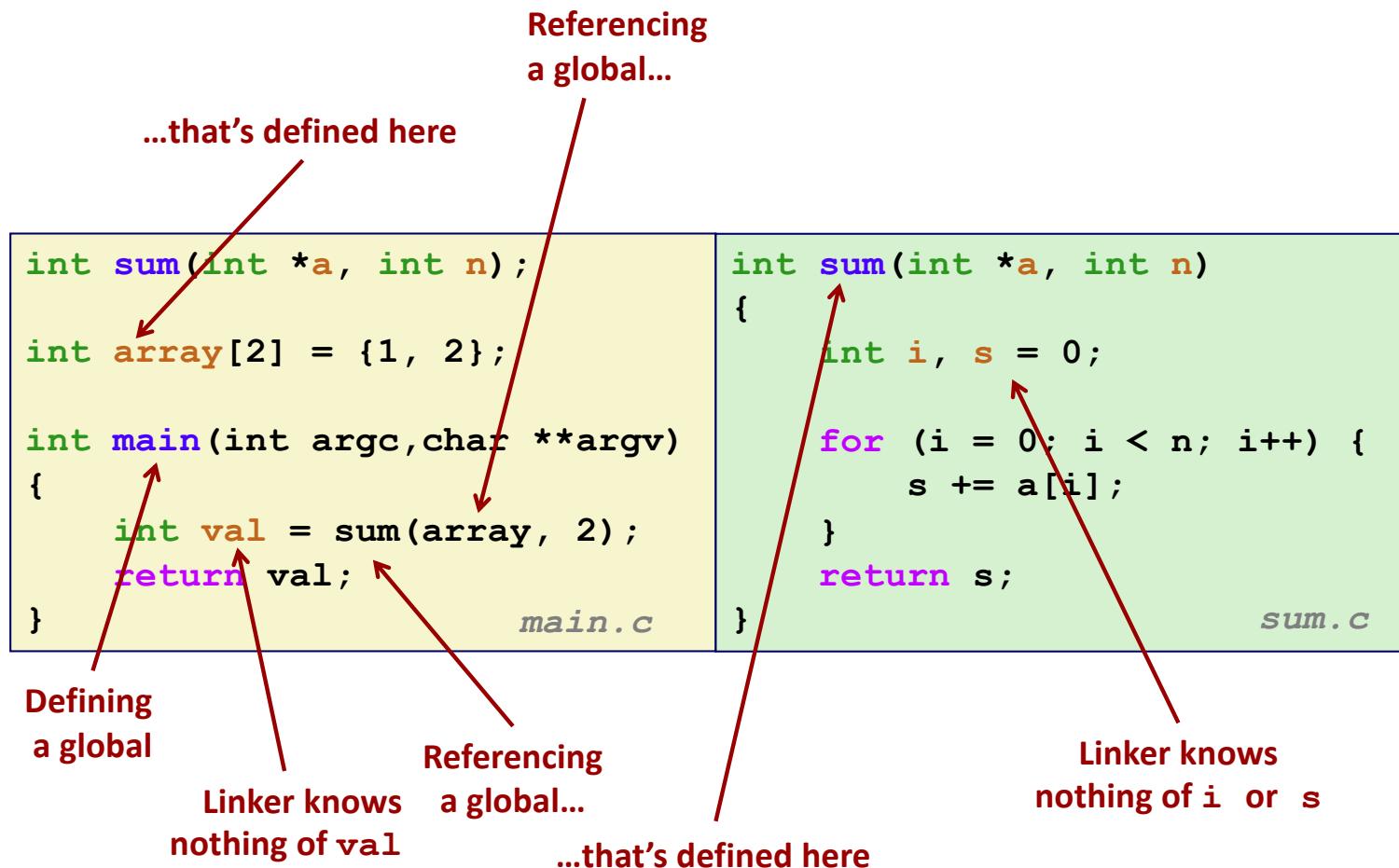
## External symbols

- Global symbols that are referenced by module  $m$  but defined by some other module.

## Local symbols

- Symbols that are defined and referenced exclusively by module  $m$ .
- E.g.: C functions and global variables defined with the **static** attribute.
- Local linker symbols are *not* local program variables**

# Step 1: Symbol Resolution



# Symbol Identification

*Which of the following names will be in the symbol table of `symbols.o`?*

(P)RIS F. H. C. S. I. R. D. H. N. V. P. C. (N)

```

symbols.c:
int time;          (Global func.)
int foo(int a) {
    int b = a + 1;
    return b;
}
int main(int argc,
         char* argv[]) {
    printf("%d\n", foo(5));
    return 0;
}
  
```

(P)RIS F. H. C. S. I. R. D. H. N. V. P. C. (N)

## Names:

- `time`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- `"%d\n"`

Can find this with `readelf`:

`linux> readelf -s symbols.o`

# Local Symbols

(ASL en-US 10.0) (Ansible, /usr/share/cross)

## ■ Local non-static C variables vs. local static C variables

- local non-static C variables: stored on the stack
- local static C variables: stored in either .bss, or .data

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

*static-local.c*

Compiler allocates space in .data for each definition of x

Creates local symbols in the symbol table with unique names, e.g., x, x.1721 and x.1724.

# Symbols and relocation

~~only~~

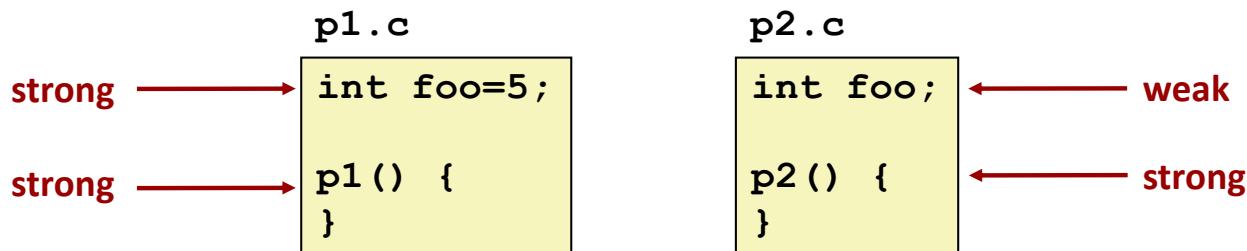
1. Symbols are compiler-generated tokens used for resolving dependencies across object files ✓
2. Relocatable code has the information for the linker to replace references to external code in another .o file ✓
3. In the executable the symbols are replaced with their absolute addresses ✓
4. Debug symbols are located inside the symbol table in ELF ✓, n ?
5. Static global variables in C cannot be accessed outside of the source file they are defined in. ✓

(.DLL -> versão versão versão, versão)

# How Linker Resolves Duplicate Symbol Definitions

## ■ Program symbols are either *strong* or *weak*

- **Strong**: procedures and initialized globals
- **Weak**: uninitialized globals
  - Or ones declared with specifier extern



# Linker's Symbol Rules

## ■ Rule 1: Multiple strong symbols are not allowed

- Each item can be defined only once
- Otherwise: Linker error

## ■ Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol

- References to the weak symbol resolve to the strong symbol

## ■ Rule 3: If there are multiple weak symbols, pick an arbitrary one

- Can override this with `gcc -fno-common`

!WTF?

## ■ Puzzles on the next slide

# Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Two type mismatch errors  
**extern int a;** : user error  
**Global variable error**  
**Link time error: two strong symbols (p1)**

(  
↳  
↳  
↳)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same uninitialized int. Is this what you really want?

```
extern
int x;
int y;
p1() {}
```

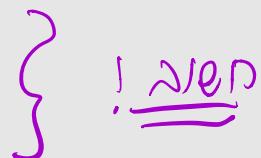
```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**!  
 Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**!  
 Nasty!



```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable.

**Important:** Linker does not do type checking.

# Type Mismatch Example

```
long int x; /* Weak symbol */

int main(int argc,
          char *argv[])
{
    printf("%ld\n", x);
    return 0;
}
```

*mismatch-main.c*

```
/* Global strong symbol */
double x = 3.14;
```

*mismatch-variable.c*

- Compiles without any errors or warnings
- What gets printed?

# Global Variables

global variables

- Avoid if you can
- Otherwise
  - Use **static** if you can
  - Initialize if you define a global variable
  - Use **extern** if you reference an external global variable
    - Treated as weak symbol
    - But also causes linker error if not defined in some file
  - Use different names

Linker : PLD'Ø

## ANSWER

(٤) المعنى: دلالة الكلمة في المقام المخصوص بها، وهي دلالة الكلمة في المقام المخصوص بها.

אנו מודים לך על תרומותך ותומךך בפזון. מילויים לך ותודה לך.

• الآن نبدأ مع الآن نبدأ مع الآن نبدأ مع الآن نبدأ مع الآن

וְאֵת הַזָּהָר יִמְצָא בְּבֵית כֹּהֵן גָּדוֹלָה וְאֵת הַזָּהָר יִמְצָא בְּבֵית שְׂדֵךְ (1)

(2) למד מילון גיבוב מילים כיון שהלן מלים מהן תנסה:

(relocatable) file has relocatable header. One segment : relocatable (2)

וְאֵין תַּחֲזִק בָּרֶךְ יְהוָה בְּבָנָיו וְאֵין תַּחֲזִק בָּנָיו בְּבָנָיו

لـ "Col" linker  $\rightarrow$  مـ "j" مـ "n" مـ "l" مـ "r" مـ "s" مـ "t" مـ "u" مـ "v" مـ "w" مـ "x" مـ "y" مـ "z"

b. s'entraîner

Linker  $\rightarrow$  PIZZ (3)

$r'$ -Symbol  $\rightarrow$   $\beta_{\text{def}}$  run in  $\beta$ : Symbol resolution : (10) Re

הנתקן בפונט מילויים (בזבז) ומיון (בזבז)

SIN'O IS negative PAST TENSE is consist of parts

, Franco Rivera Grimaldo ne wypis : Revocation : (2) sk

ا، ایج، سرے، دنیا، فی، نیز

ရှုချေးမှုပါး : a.out (2), (relocatable) .o (10) : ကိုအသွေ ပေးပို့ရန် (4)

.(P'ג(לע'ג) P'ג>QN P'ג>) Sh (d)

(ELF Container). ELF Container Definition : ELF Container (5)



# Use of `extern` in .h Files (#1)

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

```
#include <stdio.h>
#include "global.h"

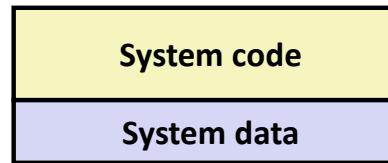
int g = 0; ← (strong)

int main(int argc, char argv[]) {
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

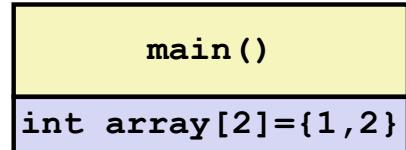
## Step 2: Relocation

רפלוקטָבָל אֹbjְּקְט פְּלִילְּסְטָן  
 (רְלֶוקְּפָּטְּ) סְקָטְּ-עַדְּמָה (רְלֶוקְּפָּטְּ)  
 (רְלֶוקְּפָּטְּ) סְקָטְּ-עַדְּמָה (רְלֶוקְּפָּטְּ)

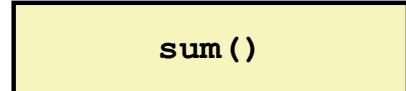
### Relocatable Object Files



main.o



sum.o



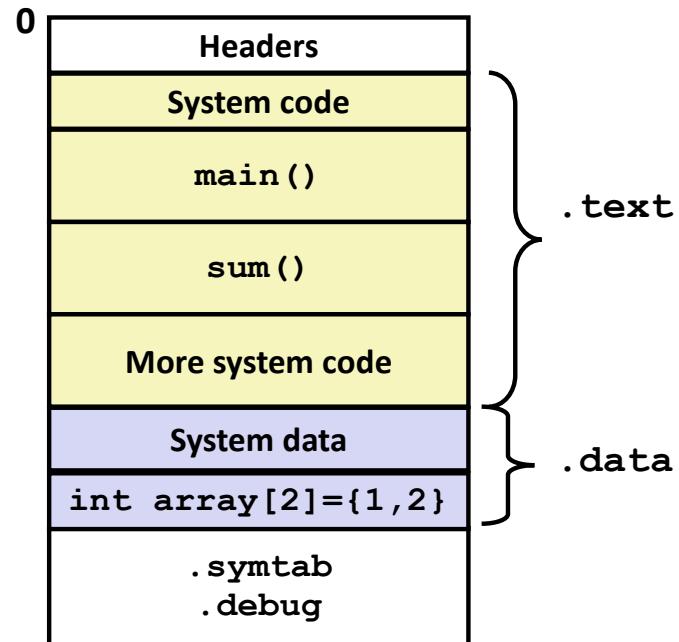
.text  
.data

.text  
.data

.text

### Executable Object File

תְּקָרְבָּנְדָּה  
 TEXT SECTION → TEXT SEGMENT



.text

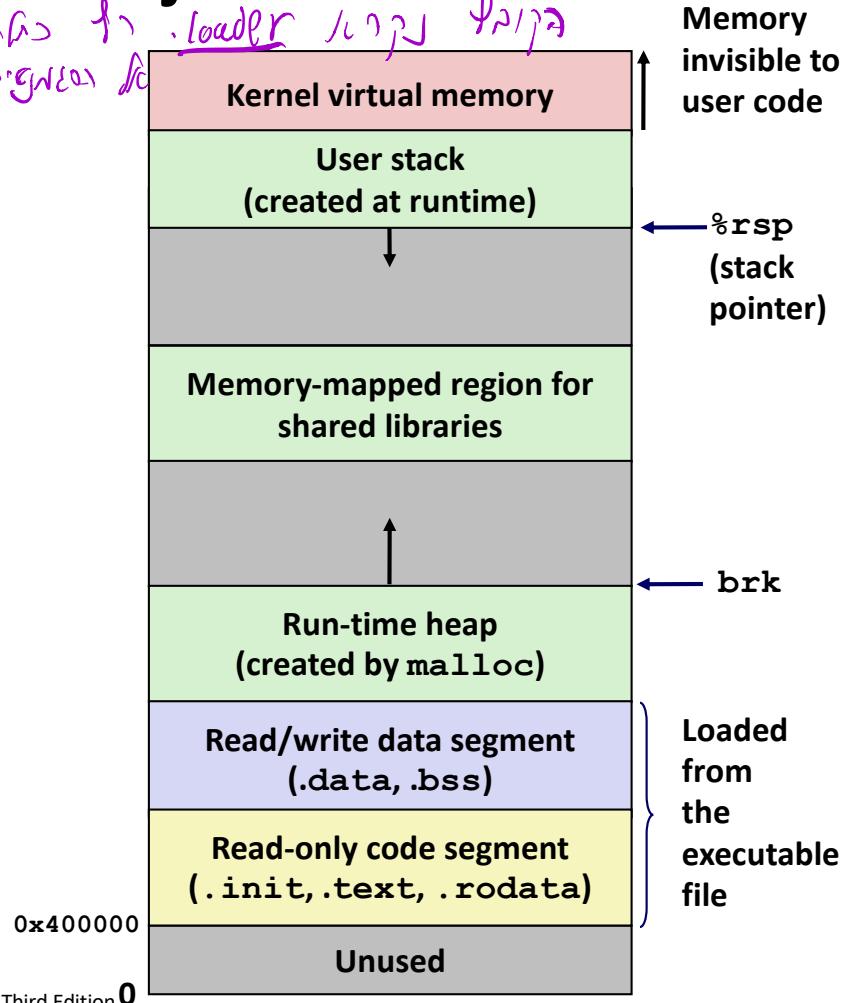
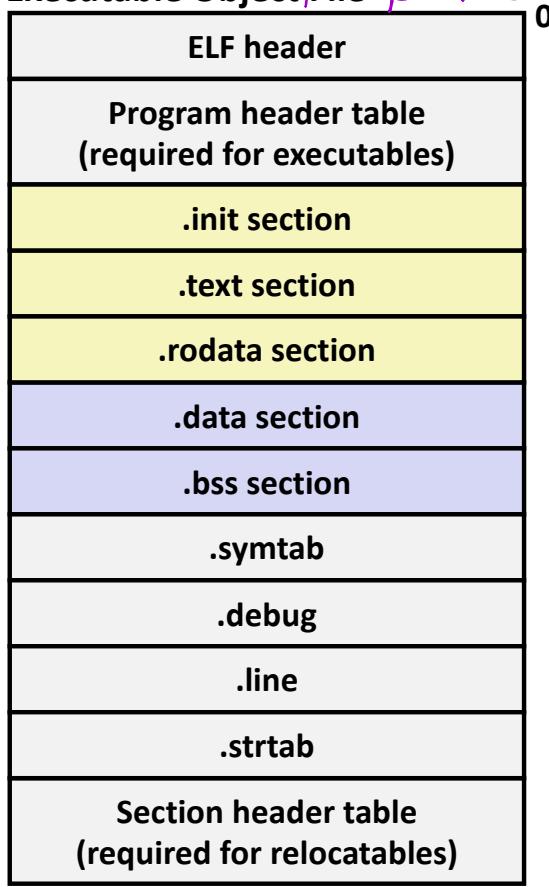
.data

רְלֶוקְּפָּטְּ

איך מפוזר פайл אקסצ'utable, איזה גורם מפוזר, layout -> מה עלה בראן

# Loading Executable Object Files

ELF -> פועל על PPC ו-ARM, מנגנון קובץ ייחודי לארכיטקטורה, Loader מפוזר פайл אקסצ'utable



## Symbols resolution

The linker will fail to link these two files defining a global variable

1. A.c: { static int i; } B.c {static int i;} f ✓ *ולפיג'ה גלוּגַגְגָה יְמִינָה שְׁמִינִית*
  2. A.c: { int i; } B.c { double i; } f *ולפיג'ה ר'הוֹגְגָה נְמִינָה בְּמִינָה וְלִימָנָה יְמִינָה שְׁמִינִית*
  3. A.c: { extern int i; } B.c { extern int i;} t *... כְּלִימָד*
  4. A.c: { int i=9; } B.c { int i=10;} t
  5. A.c: {int i=0; } B.c {extern i;} f

# Packaging Commonly Used Functions

- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.
- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

# Old-fashioned Solution: Static Libraries

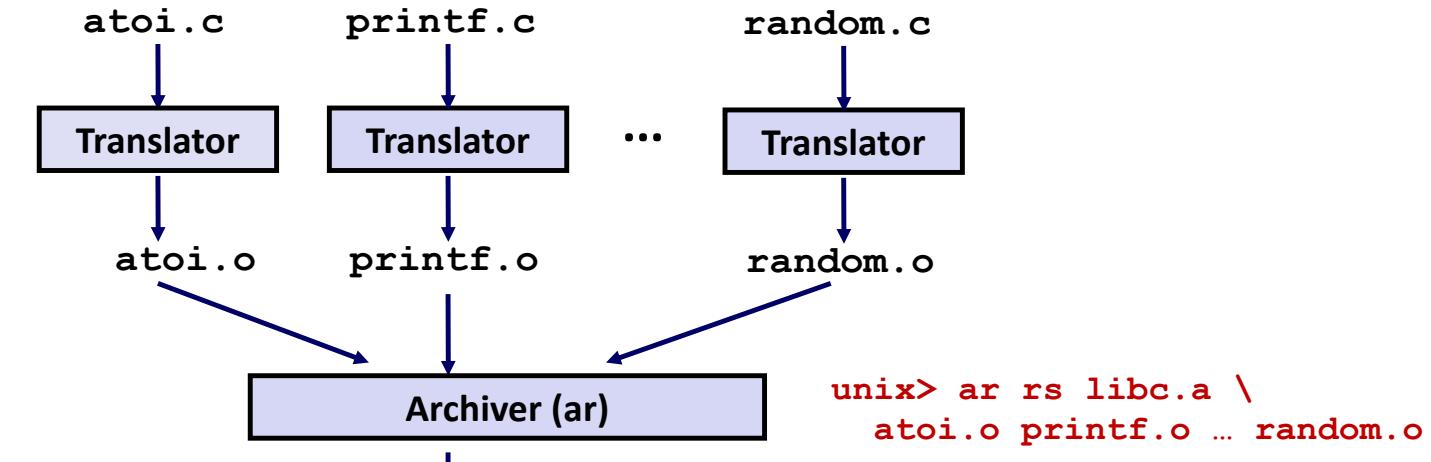
... פ'ג אַבְרָהָם סְעִירָה כְּלֵי רְמִים וְמִזְבְּחָה

## ■ Static libraries (.a archive files)

- Concatenate related relocatable object files into a single file with an index (called an *archive*).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link it into the executable.

. פ'ג אַבְרָהָם סְעִירָה כְּלֵי רְמִים וְמִזְבְּחָה :Static library

# Creating Static Libraries



unix> ar rs libc.a \  
 atoi.o printf.o ... random.o

libc.a C standard library  
.a

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

# Commonly Used Libraries

## **libc.a (the C standard library)**

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

## **libm.a (the C math library)**

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t /usr/lib/libc.a | sort
...
fork.o
...
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

```
% ar -t /usr/lib/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
...
```

# Linking with Static Libraries

```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char** argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
           z[0], z[1]);
    return 0;          main2.c
}
```

libvector.a

```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

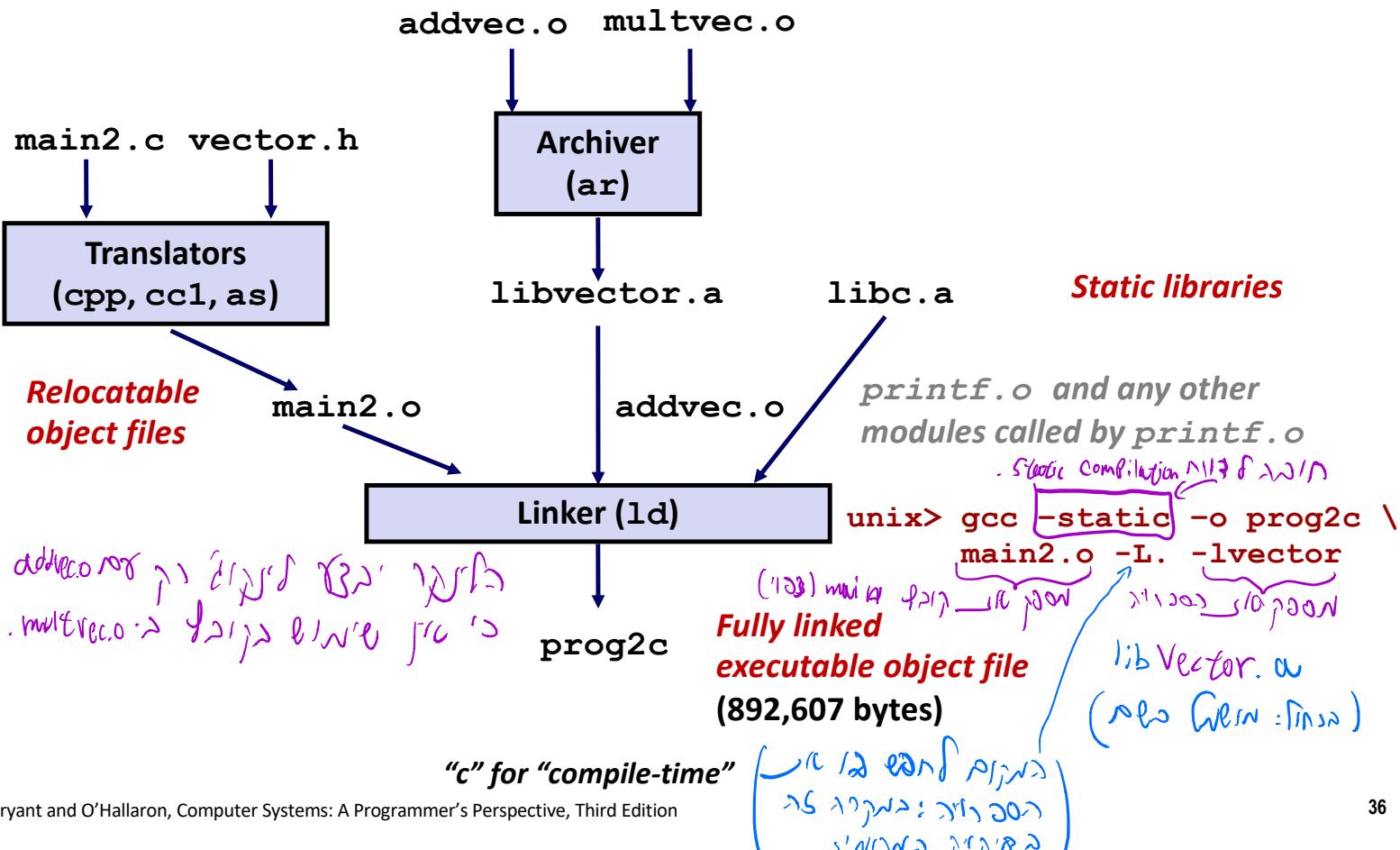
*addvec.c*

```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

*multvec.c*

# Linking with Static Libraries



# Modern Solution: Shared Libraries

## ■ Static libraries have the following disadvantages:

- Duplication in the stored executables (every function needs libc)
- Duplication in the running executables
- Minor bug fixes of system libraries require each application to explicitly relink
  - Rebuild everything with glibc?
  - <https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html>

## ■ Modern solution: Shared Libraries

- Object files that contain code and data that are loaded and linked into an application *dynamically*, at either *load-time* or *run-time*
- Also called: dynamic link libraries, DLLs, .so files

# Shared Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (ld-linux.so).
  - Standard C library (libc.so) usually dynamically linked.
- Dynamic linking can also occur after program has begun (run-time linking).
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory



# What dynamic libraries are required?

(P'-Section word)

## ■ .interp section

- Specifies the dynamic linker to use (i.e., `ld-linux.so`)

## ■ .dynamic section

- Specifies the names, etc of the dynamic libraries to use
- Follow an example of `prog`

(NEEDED)

Shared library: [libm.so.6]

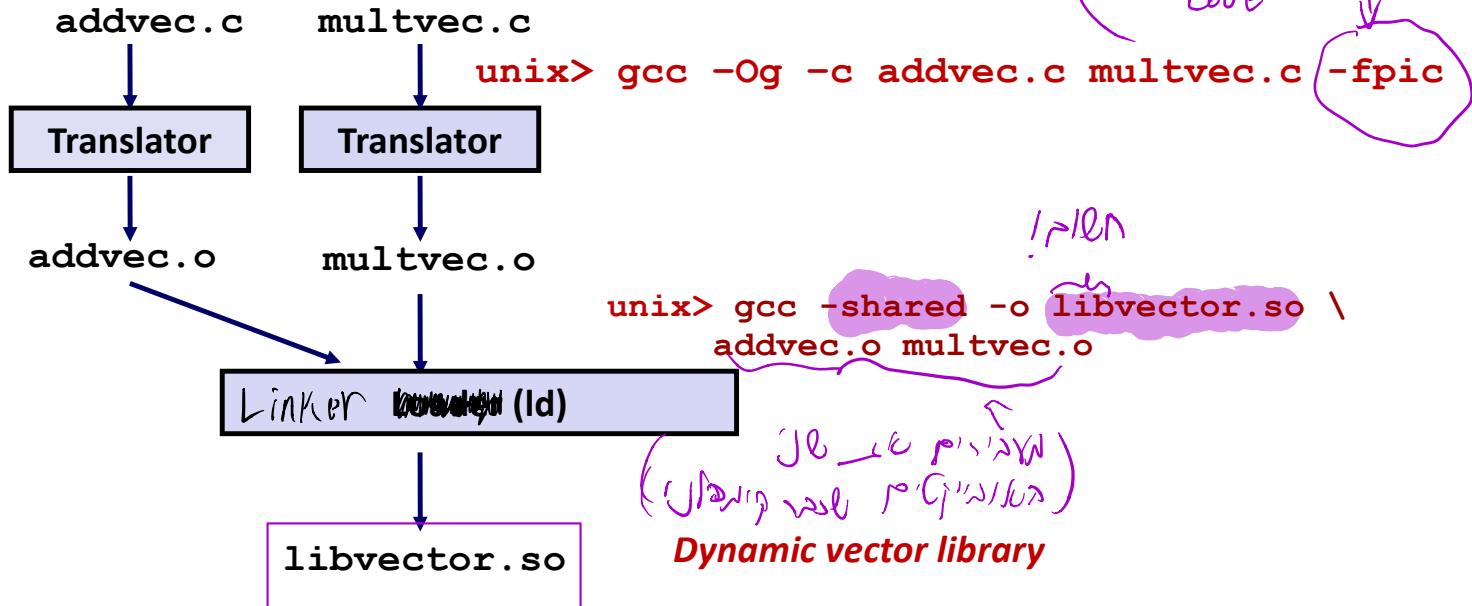
## ■ Where are the libraries found?

- Use “`ldd`” to find out:

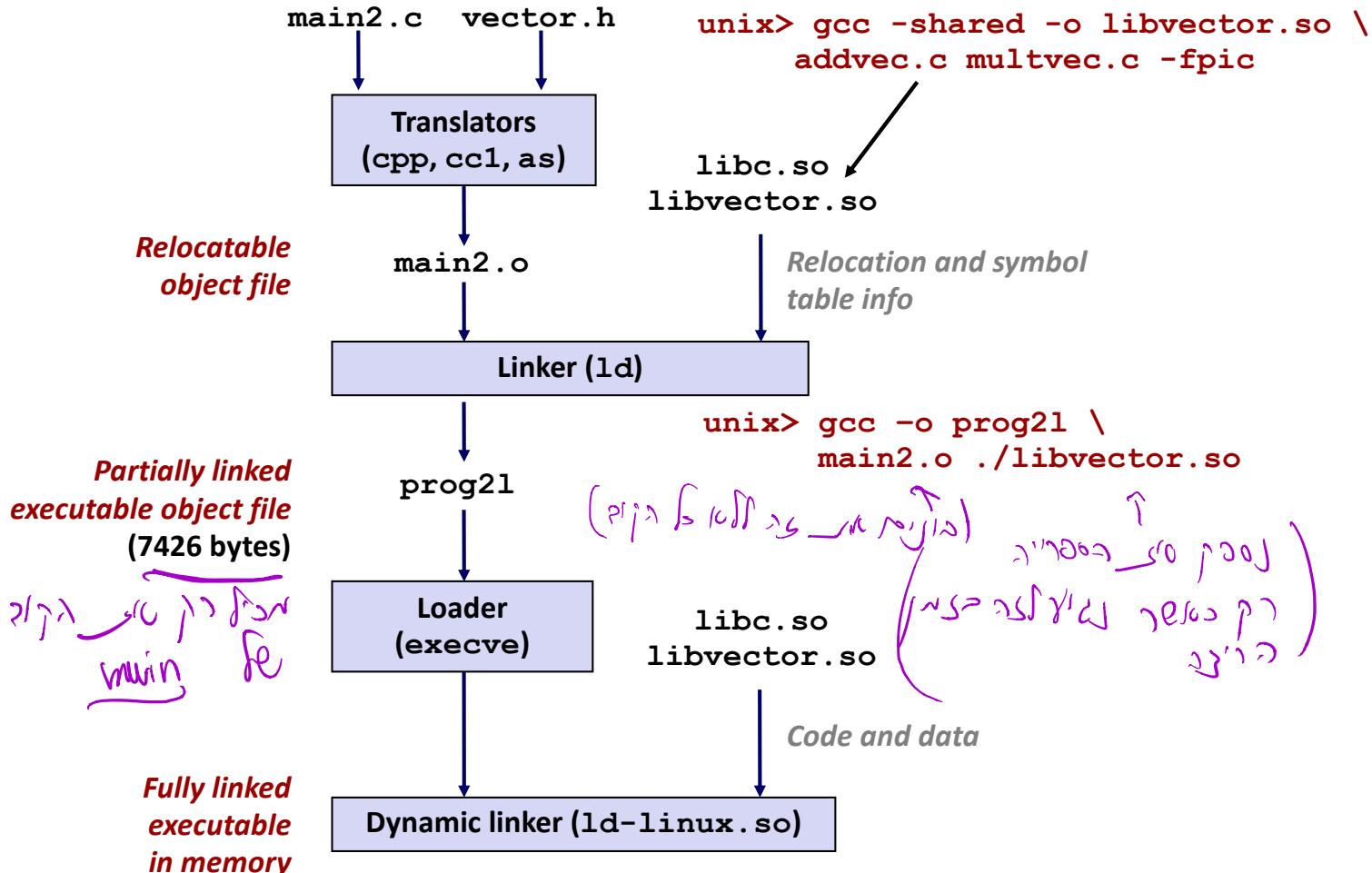
(Linux system search paths for shared libraries)  
↓  
Program Path to find the library

```
unix> ldd prog
linux-vdso.so.1 => (0x00007ffcfc2998000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
/lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

# Dynamic Library Example



# Dynamic Linking at Load-time



# Dynamic Linking at Run-time

```

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char** argv)
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    . . .

```

dll.c

*(Handle)*

*(Load Library)*

# Dynamic Linking at Run-time (cont)

```
...
/* Get a pointer to the addvec() function we just loaded */
addvec = dlsym(handle, "addvec");
if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
}

/* Now we can call addvec() just like any other function */
addvec(x, y, z, 2);
printf("z = [%d %d]\n", z[0], z[1]);

/* Unload the shared library */
if (dlclose(handle) < 0) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
}
return 0;
}
```

*dll.c*

אינטראקטיבי

רילוט כ-ELF או ELF-וּלְמָן (לע"ס נוֹטָן) הינו רילוט Section 6

(ר, ו, ב) מודול אחד יתפרק למספר Sections, ו-ELF-וּלְמָן יתפרק למספר Segments : Segment (2)

.PHDR Segment layout (3)

פ-ELF-וּלְמָן, ד-ELF-וּלְמָן, layout -> ערך header (4)  
ר-ELF-וּלְמָן Sections ELF ->

ה-ELF-וּלְמָן הוא package ו-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן (5)  
כגון object, library, dynamic linking ועוד. ה-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן.  
לכל ELF-וּלְמָן יהיה סידור ו-ELF-וּלְמָן (לע"ס סידור ו-ELF-וּלְמָן).

לכל ELF-וּלְמָן (.o) יהיה סידור ו-ELF-וּלְמָן (6) : static library  
בנוסף לכך ישנו סידור ו-ELF-וּלְמָן ש-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן  
לכל ELF-וּלְמָן (.o). גורם זה ש-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן.  
ה-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן.

ה-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן : Shared libraries (7)

ה-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן ש-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן  
לכל ELF-וּלְמָן (.so). גורם זה ש-ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן  
לכל ELF-וּלְמָן (.so).

: static library -> WTF Shared libraries -> סידור ELF-וּלְמָן (8)

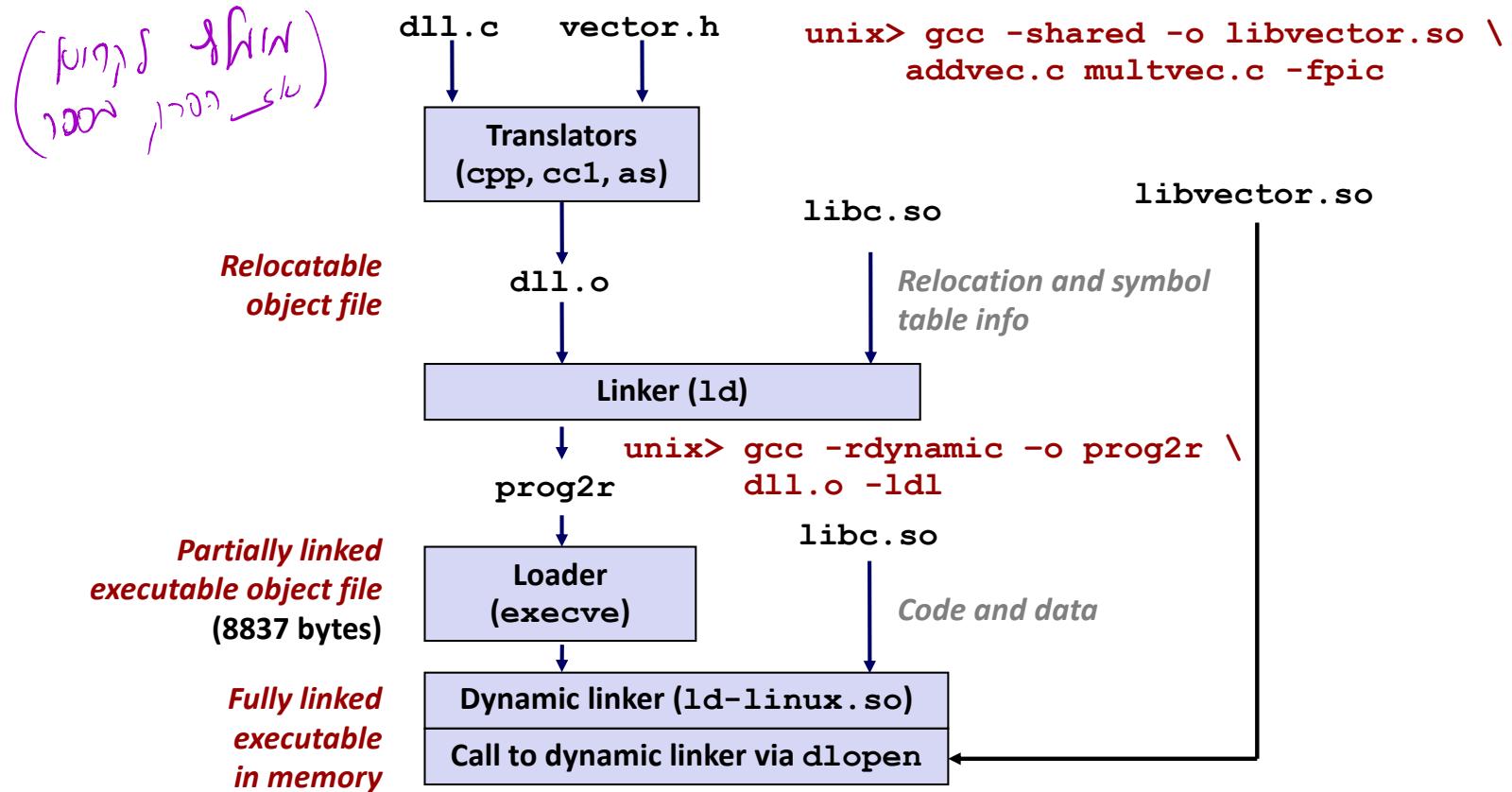
סידור ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן, ELF-וּלְמָן יתפרק ל-ELF-וּלְמָן (1)

.elf 파일 יתפרק ל-ELF-וּלְמָן Sections (2)

Sections יתפרק ל-ELF-וּלְמָן (3)

ולא יתפרק ל-ELF-וּלְמָן (4)

# Dynamic Linking at Run-time



# Linking Summary

- **Linking is a technique that allows programs to be constructed from multiple object files.**
- **Linking can happen at different times in a program's lifetime:**
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)
- **Understanding linking can help you avoid nasty errors and make you a better programmer.**

# Static/dynamic libs

1. Static libraries are just a set of object files brought together in a single file
2. Linking with static libraries copies all the functions into the target binary
  
3. Dynamic linking loads the library only at run-time
4. Shared library must be compiled into a position independent code (only relative addresses)
5. Dynamic linking does not require the library.so at link-time
6. Dynamic-linking at runtime allows the developer to load any library

# Today

- Linking
- Case study: Library interpositioning

# Case Study: Library Interpositioning

- Documented in Section 7.13 of book
- Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions
- Interpositioning can occur at:
  - Compile time: When the source code is compiled
  - Link time: When the relocatable object files are statically linked to form an executable object file
  - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

# Some Interpositioning Applications

## ■ Security

- Confinement (sandboxing)
- Behind the scenes encryption

## ■ Debugging

- In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
- Code in the SPDY networking stack was writing to the wrong location
- Solved by intercepting calls to Posix write functions (write, writev, pwrite)

Source: Facebook engineering blog post at:

<https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/>

# Some Interpositioning Applications

## ■ Monitoring and Profiling

- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
  - Detecting memory leaks
  - **Generating address traces**

## ■ Error Checking

- C Programming Lab used customized versions of malloc/free to do careful error checking

# Example program

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main(int argc,
          char *argv[])
{
    int i;
    for (i = 1; i < argc; i++) {
        void *p =
            malloc(atoi(argv[i]));
        free(p);
    }
    return(0);
}
```

int.c

- Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.
- Three solutions: interpose on the library `malloc` and `free` functions at compile time, link time, and load/run time.

# Compile-time Interpositioning

```
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

# Compile-time Interpositioning

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc intc.c mymalloc.o
linux> make runc
./intc 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
linux>
```

Search for <malloc.h> leads to  
/usr/include/malloc.h

Search for <malloc.h> leads to

# Link-time Interpositioning

```
#ifdef LINKTIME
#include <stdio.h>

void * __real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void * __wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

# Link-time Interpositioning

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl \
    int.o mymalloc.o
linux> make runl
./intl 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

Search for <malloc.h> leads to  
/usr/include/malloc.h

- The “**-Wl**” flag passes argument to linker, replacing each comma with a space.
- The “**--wrap,malloc**” arg instructs linker to resolve references in a special way:
  - Refs to malloc should be resolved as \_\_wrap\_malloc
  - Refs to \_\_real\_malloc should be resolved as malloc

# Load/Run-time Interpositioning

Observe that DON'T have  
`#include <malloc.h>`

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlsfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

# Load/Run-time Interpositioning

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

# Load/Run-time Interpositioning

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
...
linux>
```

Search for `<malloc.h>` leads to  
`/usr/include/malloc.h`

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.
- Type into (some) shells as:

```
(setenv LD_PRELOAD "./mymalloc.so"; ./intr 10 100 1000)
```

# Interpositioning Recap

## ■ Compile Time

- Apparent calls to `malloc/free` get macro-expanded into calls to `mymalloc/myfree`
- Simple approach. Must have access to source & recompile

## ■ Link Time

- Use linker trick to have special name resolutions
  - `malloc` → `__wrap_malloc`
  - `__real_malloc` → `malloc`

## ■ Load/Run Time

- Implement custom version of `malloc/free` that use dynamic linking to load library `malloc/free` under different names
- Can use with ANY dynamically linked binary

```
(setenv LD_PRELOAD "./mymalloc.so"; gcc -c int.c)
```

# Linking Recap

- **Usually: Just happens, no big deal**
- **Sometimes: Strange errors**
  - Bad symbol resolution
  - Ordering dependence of linked .o, .a, and .so files
- **For power users:**
  - Interpositioning to trace programs with & without source

# Unix I/O Overview

(פָּרָשָׁה דְּבָרִים רַקֵּן תְּלִיפָּת)

- A Linux **file** is a sequence of  $m$  bytes:

- $B_0, B_1, \dots, B_k, \dots, B_{m-1}$

לְכֹל מִזְרָחָה וְמִזְרָחָה לְכֹל כָּלָל

- Cool fact: All I/O devices are represented as files:

- `/dev/sda2` (`/usr` disk partition)
  - `/dev/tty2` (terminal)

- Even the kernel is represented as a file:

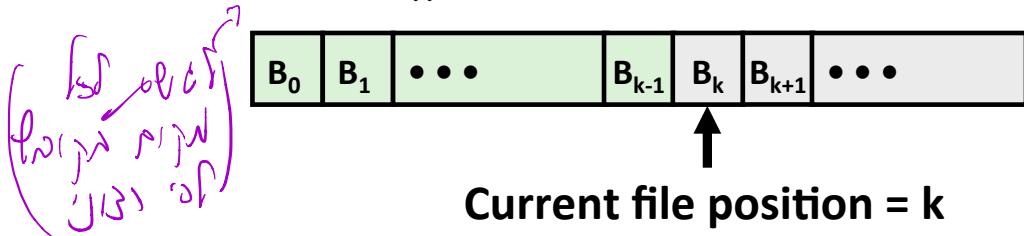
- `/boot/vmlinuz-3.13.0-55-generic` (kernel image)
  - `/proc` (kernel data structures)

(bag of bytes  $\rightarrow$   $\text{I/O}$   $\rightarrow$   $\text{CPU}$ )

# Unix I/O Overview

- Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O*:

- Opening and closing files
  - `open()` and `close()`
- Reading and writing a file
  - `read()` and `write()`
- Changing the *current file position* (seek)
  - indicates next offset into file to read or write
  - `lseek()`



# File Types

(Regular file, socket, symbolic link, character, block)

- Each file has a **type** indicating its role in the system
  - *Regular file*: Contains arbitrary data
  - *Directory*: Index for a related group of files
  - *Socket*: For communicating with a process on another machine
- Other file types beyond our scope
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

# Regular Files

логік змінні : реальність фізичні  
 інформації  
 → E/A Form Code

- A regular file contains arbitrary data
- Applications often distinguish between *text files* and *binary files*
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
    - Kernel doesn't know the difference!
- **Text file is sequence of *text lines***
  - Text line is sequence of chars terminated by *newline char* ('\n')
    - Newline is **0xa**, same as ASCII line feed character (LF)
- **End of line (EOL) indicators in other systems**
  - Linux and Mac OS: '\n' (**0xa**)
    - line feed (LF)
  - Windows and Internet protocols: '\r\n' (**0xd 0xa**)
    - Carriage return (CR) followed by line feed (LF)



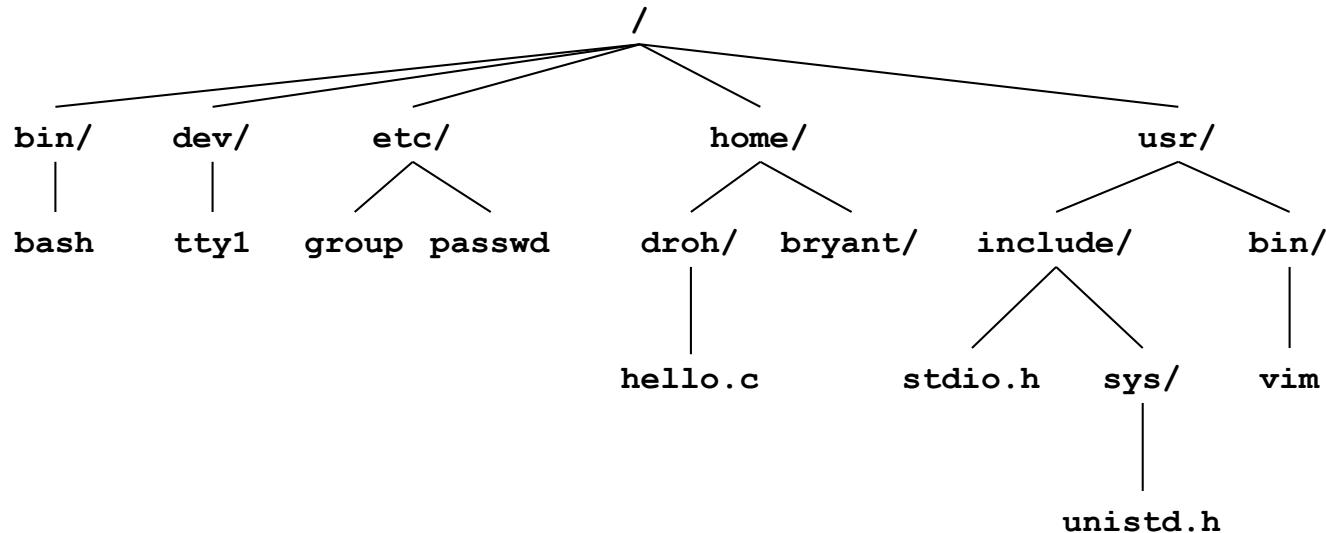
# Directories

- Directory consists of an array of *links*
  - Each link maps a *filename* to a file
- Each directory contains at least two entries
  - . (dot) is a link to itself
  - .. (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- Commands for manipulating directories
  - `mkdir`: create empty directory
  - `ls`: view directory contents
  - `rmdir`: delete empty directory

# File System Organization

## Directory Hierarchy

- All files are organized as a hierarchy anchored by root directory named / (slash)

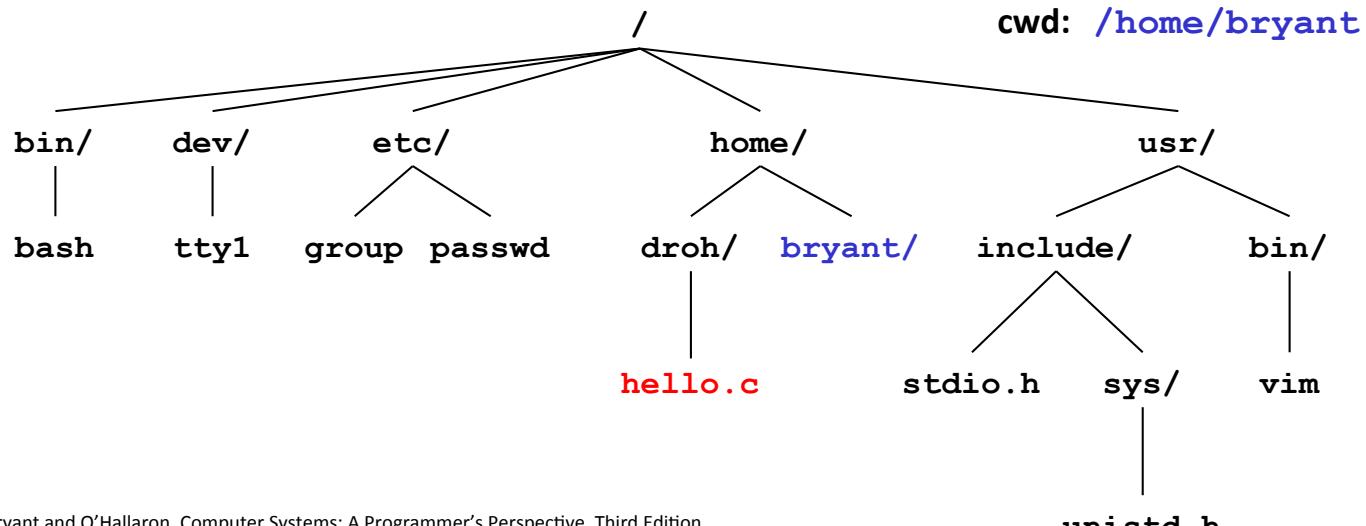


- Kernel maintains *current working directory (cwd)* for each process
  - Modified using the `cd` command

# Pathnames

## ■ Locations of files in the hierarchy denoted by *pathnames*

- *Absolute pathname* starts with '/' and denotes path from root
  - `/home/droh/hello.c`
- *Relative pathname* denotes path from current working directory
  - `../home/droh/hello.c`



# Opening Files

• چنانچه فایل برای خواندن یا نوشتن باید باز شود، مگر وسیله ای که

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd; /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- Returns a small identifying integer **file descriptor**
  - `fd == -1` indicates that an error occurred
- Each process created by a Linux shell begins life with three open files associated with a terminal:

- 0: standard input (`stdin`)
- 1: standard output (`stdout`)
- 2: standard error (`stderr`)

• چنانچه راهنمایی را داشته باشید

# Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

(لأجله ، يجب إغلاقه)  
.

- Closing an already closed file is a recipe for disaster in threaded programs (more on this later)
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

# Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```

char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}

```

Read is in  
? ↗ (P)

- Returns number of bytes read from file **fd** into **buf**

- Return type **ssize\_t** is signed integer
- **nbytes < 0** indicates that an error occurred
- **Short counts** (**nbytes < sizeof(buf)**) are possible and are not errors!

# Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;     /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

پرسیدنی را که اینجا پیش نموده است را بدانید

- Returns number of bytes written from **buf** to file **fd**
  - **nbytes** < 0 indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

# On Short Counts

## ■ Short counts can occur in these situations:

- Encountering (end-of-file) EOF on reads
- Reading text lines from a terminal
- Reading and writing network sockets

## ■ Short counts never occur in these situations:

- Reading from disk files (except for EOF)
- Writing to disk files

## ■ Best practice is to always allow for short counts.

# Files

1. Calling `read()` twice to read one byte, on the same file will read the same data ✓ (הפעלה כפולה של פונקציית הקריאה תחזיר את אותוバט)
2. `read()` cannot read the file backwards ✓ (הקריאה לאחור אינה מותרת)
3. `read()` can read data in bytes ✓ (הקריאה יכולה להיות בbytes או בoctets)
4. `open()` and `close()` are similar to “`malloc()`” and “`free()`” but just for files ✓
5. Only text files can be read/written using `read()`/`write()` ✓

(buffer ו fp) הם ר'ז'אנס מואזן open/close -> מנגנון פונקציונלי

פונקציית close מארחת את ההפניות מהfp, בכך נסגר  
free -> סגירה

## Error handling

- **Always** check return values when calling external functions
  - In Linux:

(\lambda \mu \nu \rho \gamma)

- error return value is usually “-1”
  - The reason for the error is stored in a global variable “int errno” (defined in <errno.h>)
  - Note! Successful calls may also update errno, but it should be ignored!
  - Use perror(“Error happened”) to print the reason in human readable form

(Error)  $\leftarrow$   $\text{new\_err}$   $\leftarrow$   $\text{err}$   $\oplus$   $\text{err}$ )

**Checking return values is a MUST!! Always check them!**

בנוסף ל-רפלקס פטנט (reflex patellar) ניתן לשים בדיקה נוספת על מנת לבדוק את ה-טראפזיאליים (trapzials). בבדיקה זו מושך את ה-טראפזיאליים (trapzials) ובודק אם יש תגובה של ה-טראפזיאליים (trapzials) ב-

•  $f_{\geq 1}$   $f_{\leq 1}$   $\text{map}$   $\text{S}$   $\rightarrow$   $f_{\geq 1}$   $f_{\leq 1}$   $\text{for}$   $\rightarrow \text{join}(\mathbf{x})(1)$   
• Bag of Bytes  $f_{\geq 0}$   $f_{\leq 0}$   $f_{\geq 1}$   $f_{\leq 1}$  (2)

Open, close if ~~wrong~~ (~~wrong answer~~) for trip B (c)

Write-1 reads: if  $\delta \geq \gamma$  (malloc, free) or  $\alpha \geq \gamma$  (read)  $\rightarrow$  S1

current file position is right in memory stack to  $\text{P}_0$   $\_\_r\text{pc}$

. I seek app(ja) 'f' e) 12120 Lc(ju) U's'e' M(c) 10110

(Write-read-pair-instruction) Write-read-pair (VR)

: №32) 1610 (2

(0111) You use ~~the~~ Socket (2) ~~to~~ ~~get~~ (3) file (4)

۳) نحویں دفعہ میں سے کوئی نہیں۔

\n - 2nd year in the fs, Unicode & ASCII to file per file : Copf (sc)

• ~~OK~~  $\rightarrow$  ~~final~~  $\rightarrow$  ~~OK~~

(a) jpeg (b) pdf (c) ps (d) ps (e) pdf (f) ps

الآن نحن في مرحلة الدراسات العليا وندرس كل من المفاهيم والنظريات والطرق

וְאֵלֶיךָ יִתְהַלֵּךְ כִּי־בְּעֵד־זֹאת תִּתְהַלֵּךְ וְאֵלֶיךָ יִתְהַלֵּךְ כִּי־בְּעֵד־זֹאת תִּתְהַלֵּךְ

Let's go over the following section of the code.

جف، بـلـ وـنـ سـجـلـ لـلـمـنـجـدـ

file descriptor  $\mapsto$  `open()` API  $\mapsto$  answer : file descriptor (5)

so this is what real/white kept liked. I think it's good for us to do

(close Y3AV PSYCURG) . Y3AV PSYCURG Y3AV PSYCURG fd -)

الحمد لله رب العالمين

לכידת אדריכל: לכידת גודלו (6)  
לכידת גודלו: לכידת גודלו  
לכידת גודלו: לכידת גודלו  
לכידת גודלו: לכידת גודלו  
לכידת גודלו: לכידת גודלו

לעתה נזכיר את הדרישה שפונקציית  $f$  תחזיר את המספרים  $x$  ו- $y$  במקומות  $i$  ו- $j$  בarray  $A$ . נזכיר שפונקציית  $StringIndex$  מודילה לנו את הפעולה  $\text{GetChar}(A, i)$ .

# Standard I/O Functions

- The C standard library (`libc.so`) contains a collection of higher-level **standard I/O** functions
  - Documented in Appendix B of K&R
- Examples of standard I/O functions:
  - Opening and closing files (`fopen` and `fclose`)
  - Reading and writing bytes (`fread` and `fwrite`)
  - Reading and writing text lines (`fgets` and `fputs`)
  - Formatted reading and writing (`fscanf` and `fprintf`)

# Standard I/O Streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in memory
  
- C programs begin life with three **open streams** (defined in `stdio.h`)
  - `stdin` (standard input)
  - `stdout` (standard output)
  - `stderr` (standard error)

```
#include <stdio.h>
extern FILE *stdin; /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Stream *extern FILE*

*① (from file descriptor view C file)*

*write(1, ...)*

# Buffered I/O: Motivation

- Applications often read/write one character at a time
  - `getc`, `putc`, `ungetc`
  - `gets`, `fgets`
    - Read line of text one character at a time, stopping at newline

## ■ Implementing as Unix I/O calls expensive

- `read` and `write` require Unix kernel calls
  - > 10,000 clock cycles

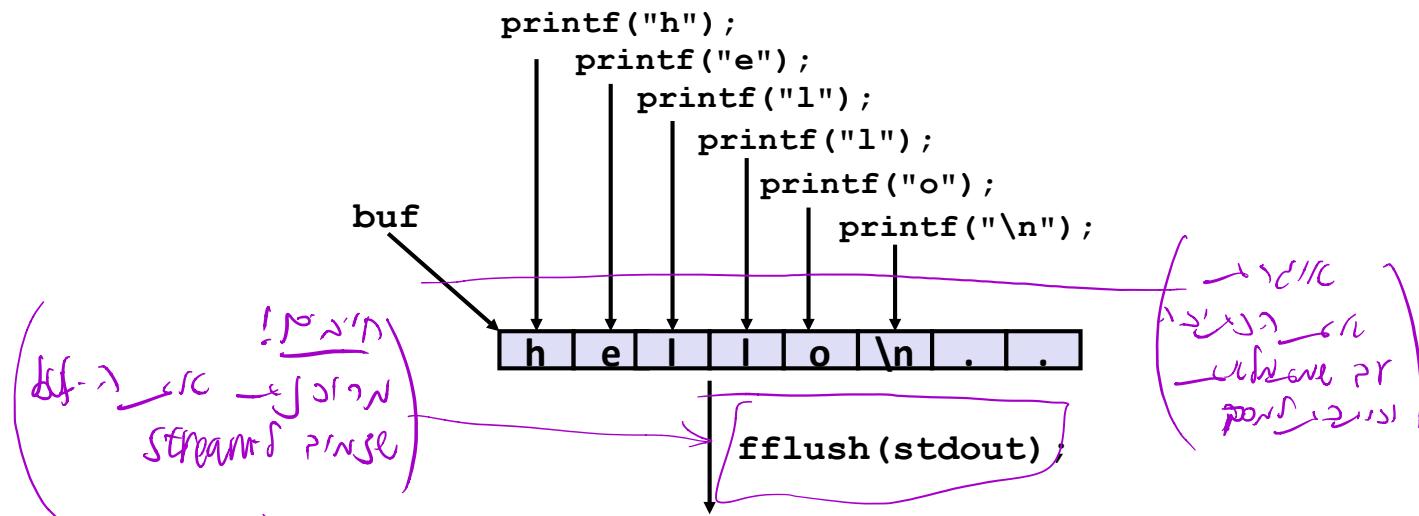
## ■ Solution: Buffered read

- Use Unix `read` to grab block of bytes
- User input functions take one byte at a time from buffer
  - Refill buffer when empty



# Buffering in Standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on "\n", call to `fflush` or `exit`, or return from `main`.

# Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

*(→ Q312N → (W) → (W) → (S) → (R) → (P) → (E) → (W))*

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

*linux> strace ./hello*

```
execve("./hello", ["hello"], /* ... */). . . = 6
write(1, "hello\n", 6) . . . = ?
exit_group(0)
```

*(↑ (S) → (E) → (D) → (O) → (U) → (C))*

*(↑ (P) → (C) → (R) → (E) → (G) → (O))*

*(↑ (R) → (L) → (E) → (N) → (H) → (O))*

*(C → P → E → W)*

*{ open  
close  
write  
read } sys calls*

# STDOUT/STDERR

Output to `stdout` is **buffered**:

Output to stderr is **not buffered** (used to print error exactly where it happens):

```
for (int i=0; i<5; i++) {
```

```
fprintf(stdout,"%d ", i);
```

```
fprintf(stderr, "%d ", i+5);
```

3

# STDOUT/STDERR

Output to stdout is **buffered**:

Output to stderr is **not buffered** (used to print error exactly where it happens):

```
for (int i=0;i<5;i++) {  
  
    fprintf(stdout,"%d ", i);  
  
    fprintf(stderr,"%d ", i+5);
```

}

(C:\> bin\>)

5 6 7 8 9 0 1 2 3 4

# Redirection in shell

- Pipe: **stdout** of one program is redirected as **stdin** for another one

```
> cat file.txt
```

12345

```
> cat file.txt | grep "hello"
```

```
> cat file.txt | grep "1"
```

12345

```
> cat file1.txt | grep "1"
```

cat: file1.txt: No such file or directory

# Redirection in shell

- > Opens a new file and writes **stdout** into it

cat file.txt > new\_file.txt: copies file.txt into new\_file.txt  
*adds*

- >> Opens an **existing file** and ~~writes~~ **stdout** into it

cat file.txt > new\_file.txt

cat file.txt >> new\_file.txt

- < Opens an existing file and writes it into **stdin**

cat file.txt | wc is the same as wc < file.txt

# Redirecting stderr into stdout

```
> cat file.txt | grep 1
```

12345

```
> cat file1.txt | grep 1
```

cat:file1.txt: no such file or directory

```
> cat file1.txt 2>&1 | grep 1
```

(Redirection  
  Stderr → Stdout)

# Redirecting to nowhere

It is often necessary to inhibit any output from a program

Use /dev/null! Ultimate black hole

cat file.txt > /dev/null

(*Copy&Run*)

Q. 15:

C file operations are mainly fixed : Libc (1)  
→ Major file interface is Libc → I/O API  
• (read, write, readline, etc.)

Streams (buffered) I/O uses buffer or we use streams (2)  
↳ for read, write, readline, etc., methods provided by  
• C I/O (2)

I/O buffering (3) , flush function : buffer (3)  
→ (0) if buf → file stderr-f ) fflush() operation  
→ (1) if no file fflush → entire buffer will print (\*)  
buf → file | if buf

# Abstract data types

Slides partially adopted from 234124. Authored by: *Gill Barequet, Gershon Elber, Omer Strulovich and Ron Rubinstein, Chaim Gotsman, Yechiel Kimchi, Yossi Gil, Eliezer Kantorowitz, Ayal Itzkoviz, Dani Kotlar and Itai Sharon, Mark Silberstein.*

# Building complex systems requires simple abstractions

*“... the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.”*

- Edsger W. Dijkstra, "The Humble Programmer" (1972)

# Abstractions = building blocks

If you want to build a car, you don't want to think about atoms in the alloy it is made of!

Instead we think about **design**: wheels, steering wheel, chassis, engine, breaks

- Each abstraction implements complex high-level function
- **Encapsulation:** Do not care/do not know/should not know how they work
- **Interface:** We know how to interact with them

# Examples of abstractions

- CPU Instruction Set Architecture
  - Interface: instructions
  - Purpose: invoke specific operation on a CPU
- Stack
  - Interface: pop/push
  - Purpose: first-in->last-out
- A programming language
  - Interface: keywords
- File
  - Interface: open/read/write/close
  - Purpose: access persistent data as a stream of bytes

Today we will talk about how to build software abstractions

# Abstract data types

- Building blocks from which we build more complex programs
  - Implement reusable software modules
- 
- How to create such a building block in C?

structs

# Motivation

- ▶ Assume we need **complex numbers** in our software
  - ◆ We create the following struct:

```
typedef struct {  
    double real;  
    double imag;  
} Complex;
```

- ▶ Is this enough?

- ◆ **No! Why?**

(*size mismatch*)

# Motivation

- ▶ What do we want from our complex type?
  - ◆ **Represent** complex numbers in our code
  - ◆ Provide ways to **use** and **operate** on these complex numbers
- ▶ While also (true for any new type, in fact):
  - (↑  
↑  
↑)
    - ◆ Prevent **code duplication**
    - ◆ **Reuse** the code we write
    - ◆ Minimize **dependency** of other code on implementation details

# Data Types

- ▶ We want a complete **data type** to represent complex numbers
  - ◆ Provide functions allowing the user to handle this type:
    - complexAdd
    - complexMultiply
    - complexAbs
    - complexPrint
    - more...

*Who is this  
user?*

DAN

**Data Type = Representation + Operations**

# Data Types

- ▶ We want our complex data type to be **reusable**
- ▶ Can be used in different files
  - Toward this end, we define **complex.h** containing the **interface** of the complex numbers data type
- ▶ Can be passed from one project to the next
  - The **implementation** should be in its own source file:  
**complex.c**

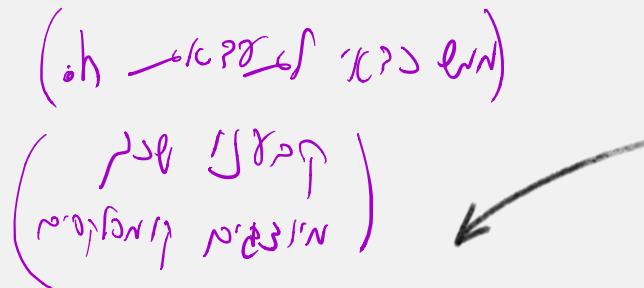
( $\text{r} + \text{i}\text{p}$ )  
 $\text{r} + \text{i}\text{p}$

# complex.h

```
#ifndef COMPLEX_H  
#define COMPLEX_H
```

```
typedef struct {  
    double real;  
    double imag;  
} Complex;
```

```
Complex complexCreate(double real, double imag);  
Complex complexCreateFromPolar(double absVal, double arg);  
int complexToString(Complex c, char buffer[], int size);  
Complex complexAdd(Complex c1, Complex c2);  
Complex complexSubtract(Complex c1, Complex c2);  
Complex complexMultiply(Complex c1, Complex c2);  
Complex complexDivide(Complex c1, Complex c2);  
Complex complexConjugate(Complex c);  
double complexAbs(Complex c);  
double complexArg(Complex c);  
  
#endif /* COMPLEX_H */
```



In general, users will have no access to the C sources.

Hence, consider documenting header files (**no room on slide...**)

# complex.c

- The .c file contains the implementation of the functions of complex

```
#include "complex.h"
#include <math.h>

Complex complexCreate(double real, double imag) {
    Complex result = { real, imag };
    return result;
}

Complex complexAdd(Complex c1, Complex c2) {
    return complexCreate(c1.real + c2.real, c1.imag + c2.imag);
}

Complex complexMultiply(Complex c1, Complex c2) {
    double a = c1.real;
    double b = c1.imag;
    double c = c2.real;
    double d = c2.imag;
    return complexCreate(a * c - b * d, b * c + a * d);
}

...
```

complex.c

# complex.c

(even though it's not clean)

- The .c file contains the implementation of the functions of complex

```
Complex complexConjugate(Complex c) {
    return complexCreate(c.real, -c.imag);
}

double complexArg(Complex c) {
    return atan2(c.imag, c.real);
}

Complex complexInverse(Complex c) {
    double norm = complexAbs(c);
    return complexMultiply(complexConjugate(c),
                           complexCreate(1.0 / square(norm), 0.0));
}

Complex complexDivide(Complex c1, Complex c2) {
    return complexMultiply(c1, complexInverse(c2));
}
```

complex.c

again, just a part of complex.c

...

# complex.c

- ▶ Some helper functions should not be available to the user
- ▶ Such functions should be declared as **static**
  - ◆ A reminder: static functions are only recognized in the file they are defined in, and do not participate in the linking phase
  - ◆ Do not declare static functions in the header file

```
static double square(double x) {
    return x * x;
}

double complexAbs(Complex c) {
    return sqrt(square(c.real) + square(c.imag));
}
```

complex.c

Why not?

What about variables?

# Benefits of Data Types

('ਵਿਧ ਪ੍ਰਕਾਸ਼ ਹੈਂਡ)

- ▶ **Reusable** - Complex is a separate module
  - ◆ Can be used in future projects conveniently
  - ◆ In fact, we might be able to find existing code implementing this data type, and reuse someone else's code
- ▶ Keeps code simple
  - ◆ The user of a complex number is spared of technical details
  - ◆ Prevents code duplication
- ▶ What about minimizing the **dependency of other code** on the implementation of Complex?

# A Dependency Problem

- ▶ Complex numbers can also be represented in **polar form**:

$$a + bi \leftrightarrow r \cdot \text{cis}(\theta)$$

- ✓ Fast multiplication
- ✓ Used for calculating the power of a number
- ✓ etc.

- ▶ Can we switch our implementation to use the polar form?

```
typedef struct {  
    double abs, arg;  
} Complex;
```

```
Complex complexMultiply(Complex c1, Complex c2) {  
    return complexCreateFromPolar(c1.abs * c2.abs, c1.arg + c2.arg);  
}
```

*So far so good...*

# A Dependency Problem

- ▶ User code of Complex **may break** if it assumes a specific implementation:

```
void print_complex(Complex c) {  
    printf("%f+%fi", c.real, c.imag);  
}
```

- ▶ How can we prevent the user from writing such implementation-specific code?
  - ◆ Solution: hide the implementation from the user completely

# Encapsulation

. struct ~~type~~ ~~the behaviour~~ ~~you~~

- ▶ We wish to prevent users of type Complex from **accessing the struct fields directly**
- ▶ We will **encapsulate**\* the implementation of the data type
  - ◆ The implementation will be inaccessible to the user
  - ◆ The type can be **manipulated through the provided interface** only

\* *Encapsulate = enclose in a capsule, seal off*

# Abstract Data Types

- ▶ C allows to define a pointer to an **incomplete type**
- ▶ User code can **pass the pointer** to functions, but cannot **access** the underlying object:

```
#ifndef COMPLEX_H_
#define COMPLEX_H_
STRUCT complex;
typedef (STRUCT complex)* Complex; // Complex is now a pointer type
// User can define a Complex variable
// But NOT a struct complex variable
```

complex.h

```
Pointers) Complex complexCreate(double real, double imag);
Complex complexCreateFromPolar(double absVal, double arg);
int complexToString(Complex c, char buffer[], int size);
Complex complexAdd(Complex c1, Complex c2);
...
#endif
```

What is the size  
of Complex?

(20) - Q10 R 1012

# Abstract Data Types

complex.c

```
#include "complex.h"

struct complex {      // The type struct complex and its fields
    double real;        // can only be used in this file
    double imag;
};

Complex complexCreate(double real, double imag) {
    Complex result = malloc(sizeof(complex));
    if (result == NULL)
        return NULL;

    result->real = real;
    result->imag = imag;
    return result;
}

...
```

# Abstract Data Types

main.c

```
#include "complex.h"

// This file can only define Complex pointers
// and pass them to the appropriate functions
```

```
int main() {
    Complex c = complexCreate(1.0, 1.0); // C is a pointer
    c->real = 0.0; // Error!
    double r = complexAbs(c);

    complexDestroy(c);
    return 0;
}
```

*don't forget!  
create => destroy...*

STRUCT complex

# Abstract Data Types

- ▶ An ADT separates the **interface** from the **implementation**
  - ◆ Using a pointer to an incomplete type gives us an **ADT in C** code
- ▶ For the user to access the real part, this must now be provided by the interface:

```
#include "complex.h"                                complex.c

...
double complexGetReal(Complex c) {                  }
    return c->real;
}

void complexSetReal(Complex c, double real) {        }
    c->real = real;
}
```

# Abstract Data Types

- ▶ An ADT defines only the **operations** (the interface) that the type supports

לעדי שטח וסימון גוף אוניברסיטאי, גודל מ'ה  
ר'ען ר'שׁוֹת א' פ'ל / ע'ו'ו'ו'ן ר'ל

- ◆ The underlying representation and implementation are unknown
- ◆ There can be several different implementations of the same ADT

**Abstract Data Type = Operations**

# Poll ADT

1. This statement will not compile:

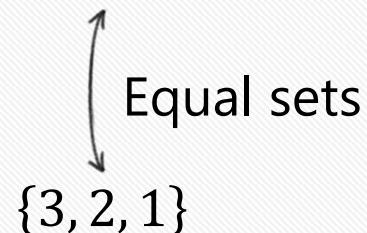
(*ר'פ'ט י'ת' ו'ג'*)  
typedef struct adt my\_adt;

typedef struct adt\* my\_adt; ✓  
typedef struct adt\*\* my\_adt; ✓

- (*ר'פ'ט י'ת' ו'ג'*)  
ADT user bus. If you are trying to hide the source code, you can replace the pointer to the struct with a pointer to a function.
2. Encapsulation is needed to protect software algorithms from theft X
  3. Pointer to a struct helps achieve encapsulation ✓ (*ר'פ'ט י'ת' ו'ג'*)
  4. ADT packaged in a shared library helps replace implementations at runtime ✓
  5. .h files are used to include the interface to an ADT ✓ (*ר'פ'ט י'ת' ו'ג'*)
  6. An ADT can only be defined in a single source file X (*ר'פ'ט י'ת' ו'ג'*) (erroneous)
  7. static functions are needed to maintain encapsulation ✓ (*ר'פ'ט י'ת' ו'ג'*)

# A Set of Integers

- ▶ A set is a collection of elements, such that no duplications are allowed
  - ◆ The main use of a set is to query whether **an element  $a$  is in the set?**
  - ◆ The **order** of the elements is **not important**
- ▶ What is the interface of a set (minimum):  $\{1, 2, 3\}$ 
  - ◆ Create              ◆ Contains
  - ◆ Destroy            ◆ Size
  - ◆ Add
  - ◆ Remove
- ▶ For now we consider a set of **integers** only (this will change...)



# A Set of Integers

- We start by defining the interface:

```
#ifndef SET_H_
#define SET_H_

#include <stdbool.h>

/** A set of integers */
typedef struct set* Set;

/** Type used for reporting errors in Set */
typedef enum {
    SET_SUCCESS, SET_OUT_OF_MEMORY, SET_NULL_ARG
} SetResult;

Set setCreate();
Set setCopy(Set set);
void setDestroy(Set set); → free
SetResult setAdd(Set set, int number);
SetResult setRemove(Set set, int number);
bool setContains(Set set, int number);
int setGetSize(Set set);
void setPrintf(Set set);

#endif
```

set.h

Should be documented!  
(No room)

(For now we will just)

These should exist for most  
ADTs



→ malloc

→ free

(we could do it for many)  
but

Useful for  
debugging

# Using the Set

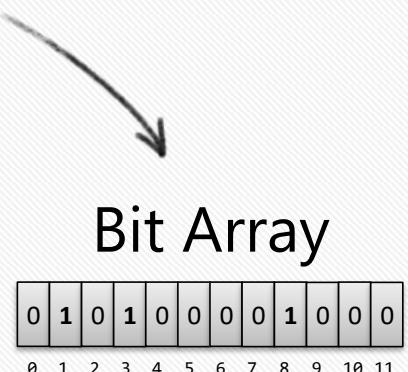
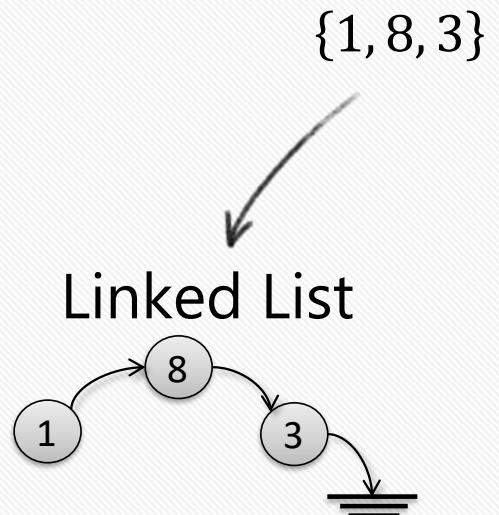
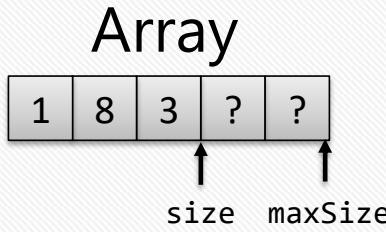
set\_app.c

```
#include "set.h"
#include <stdio.h>
(מען יתאפשר ביצוע הוראות קיימות)
int main() {
    Set set = setCreate();
    for (int j = 0; j < 20; j += 2) {
        setAdd(set, j); // should check return value!
    }
    int n;
    scanf("%d", &n);
    printf("%d is %sin the set\n", n, setContains(set, n) ? "" : "not ");
    setPrint(set);
    setDestroy(set);
    return 0;
}
```

# Implementing a Set

- ▶ How can we implement our set?

- ◆ An array
  - ◆ Perhaps keep it sorted?
- ◆ A linked list
- ◆ A bit-array



# set.c

Partial code,  
full version on the course site

if an #include is  
only needed for the  
implementation,  
put it here

**private**  
constants

```
#include "set.h"
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

/* The factor by which the set array is expanded */
#define EXPAND_FACTOR 2

/* The initial size of the set array */
#define INITIAL_SIZE 10

/* Used by 'find' for reporting a number that does not exist in the set */
#define NUMBER_NOT_FOUND -1

/* Struct representing a set implemented as an array */
struct set {
    int* elements;
    int size;
    int maxSize;
};
```

# set.c

Remember that Set  
is a **pointer** to a  
**struct set**

```
Set setCreate() {
    Set set = malloc(sizeof(set));
    if (set == NULL) {
        return NULL;
    }

    set->elements = malloc(INITIAL_SIZE * sizeof(int));
    if (set->elements == NULL) {
        free(set);
        return NULL;
    }

    set->size = 0;
    set->maxSize = INITIAL_SIZE;
    return set;
}
```

set is already  
allocated!



# set.c

```
void setDestroy(Set set) {
    if (set == NULL) {
        return;
    }
    free(set->elements);
    free(set);
}
```

Set:NULL  
elem->next  
Seg. fault

# set.c

(�්‍රිං)  
no logical return value  
in case set is NULL, so  
this is **not supported**

```
bool setContains(Set set, int number) {  
    assert(set != NULL); (අනුමත තීව්‍ය සැපයුම්)  
    for (int i = 0; i < set->size; i++) {  
        if (set->elements[i] == number) {  
            return true;  
        }  
    }  
    return false;  
}
```

our assumptions is **indicated**  
**in the code** using assert()

- What will happen in release mode?
- What are our other alternatives here?

# set.c

no need to assert, we can handle the error

why is this a success?

(copy) new use set)

a good implementation should also **shrink** the array

```
SetResult setRemove(Set set, int number) {  
    if (set == NULL) {  
        return SET_NULL_ARG;  
    }  
  
    int index = find(set, number);  
    if (index == NUMBER_NOT_FOUND) {  
        return SET_SUCCESS;  
    }  
  
    set->elements[index] = set->elements[set->size - 1];  
    set->size--;  
    return SET_SUCCESS;  
}
```

we already implemented something very similar!

(copy) new use set)

# set.c

(Implementation)

```
static int find(Set set, int number) {  
    assert(set != NULL);  
    for (int i = 0; i < set->size; i++) {  
        if (set->elements[i] == number) {  
            return i;  
        }  
    }  
    return NUMBER_NOT_FOUND;  
}
```

re-written to  
avoid code  
duplication

```
bool setContains(Set set, int number) {  
    return find(set, number) != NUMBER_NOT_FOUND;  
}
```

# set.c

```
SetResult setAdd(Set set, int number) {  
    if (set == NULL) {  
        return SET_NULL_ARG;  
    }  
  
    if (setContains(set, number)) {  
        return SET_SUCCESS;  
    }  
  
    if (set->size == set->maxSize) {  
        if (expand(set) == SET_OUT_OF_MEMORY) {  
            return SET_OUT_OF_MEMORY;  
        }  
    }  
    set->elements[set->size++] = number;  
    return SET_SUCCESS;  
}
```

expand() should  
not harm the  
set in case of  
failure

# set.c

```
static SetResult expand(Set set) {  
    assert(set != NULL);  
  
    int newSize = EXPAND_FACTOR * set->maxSize;  
    int* newElements = realloc(set->elements, newSize * sizeof(int));  
    if (newElements == NULL) {  
        return SET_OUT_OF_MEMORY;  
    }  
  
    set->elements = newElements;  
    set->maxSize = newSize;  
  
    return SET_SUCCESS;  
}
```

keep old array in  
case of failure

# Enumeration of a Set's Elements

- ▶ The user of set would probably want to iterate over its elements
  - ◆ e.g. “for each student, calculate his average”

set\_app.c

```
#include "set.h"
#include <stdio.h>

int main() {
    Set set1 = setCreate();
    for (int j = 0; j < 20; j += 2) {
        setAdd(set1, j);
    }

    int sum = 0;
    SET_FOREACH(n, set1) {
        sum += n;
    }

    printf("Sum of set is: %d\n", sum);
    setDestroy(set1);
    return 0;
}
```

Iterating over the set's  
elements  
should be supported

# Adding basic iterator support

- We will add an **internal iterator** to the set ADT to enable simple iteration over all its elements

**downside:** cannot maintain more than one iterator at a time.

**Why a problem?**

set.h

```
int setGetFirst(Set set);
bool setIsDone(Set set);
int setGetNext(Set set);

/** Macro to enable simple iteration */

#define SET_FOREACH(element, set) \
    for(int element = setGetFirst(set); \
        !setIsDone(set); \
        element = setGetNext(set))
```

set.c

```
struct set {
    int* elements;
    int size;
    int maxSize;
    int iterator;
};

int setGetFirst(Set set) {
    set->iterator = 0;
    return set->elements[0];
}

bool setIsDone(Set set) {
    return set->iterator >= set->size;
}

int setGetNext(Set set) {
    assert(!setIsDone(set));
    return set->elements[+set->iterator];
}
```

# Intro to C++

Part 1

# C++ overview

Extensions to C to improve known weaknesses

Allows programs that follow object-oriented design

Provides convenient standard libraries that offer numerous useful functions

**We will learn only the basics, mostly in the context of ADTs**

**For those interested in Object-Oriented Design, take OOP: 046271**

# Complex ADT - new version: complex.h

```
typedef struct complex * Complex;

Complex complexCreate(float re, float im);

//duplication
Complex complexClone(Complex c);

void complexDestroy(Complex c);

// assignment
void complexCopy(Complex dst, Complex src);

void complexPrint(Complex c);
```

# Complex ADT - new version: complex.c

```
#include "complex.h" // all the interfaces

struct complex{
    float re; float im;
}

static void init(Complex c, float re, float im) {c.re=re;c.im=im; }

Complex complexCreate(float re, float im) { Complex tmp=malloc(sizeof(struct complex));
                                              init(tmp,re,im); }

Complex complexClone(Complex c) { return complexCreate(c.re,c.im); }

void complexDestroy(Complex c){ free(c); }

void complexCopy(Complex src, Complex dst){ init(dst,src.re,src.im); }

void complexPrint(Complex c) { printf("%f %f\n",c.re,c.im); }
```

( !רְגַנְּרָטִוָּן בָּאַבָּד )

# How do we use it?

```
#include "complex.h" // all the interfaces
int main(){
    Complex a = complexCreate(1.0,0.0); // instance of ADT
    Complex b = complexCreate(2.0,0.0); // another instance
    complexCopy(a,b);
    // b=a; this is a bug!
    complexPrint(a);
    complexPrint(b);
    complexDestroy(a);complexDestroy(b);
    return 0;
}
```

We will call it an “object” of type  
**Complex**

# A few inconveniences

```
struct complex {  
    float re, im;  
}  
  
typedef struct complex Complex;  
  
void init(Complex *c, float re, float im) { c->re=re; c->im=im; }  
  
Complex complexCreate(float re, float im) { Complex tmp=malloc(sizeof(struct complex));  
    init(tmp,re,im); }  
  
Complex complexClone(Complex c) { return complexCreate(c->re,c->im); }  
  
void complexDestroy(Complex c){ free(c); }  
  
void complexCopy(Complex dst, Complex src){ init(dst,src->re,src->im); }  
  
void complexPrint(Complex c) { printf("%f %f\n",c->re,c->im); }
```

awkward naming that must include “complex” prefix

Must be allocated on heap to allow encapsulation

awkward naming that must include “complex” prefix

# A few inconveniences

```
struct complex{  
    float re; float im;  
}  
typedef struct complex * Complex;  
  
void init(Complex c, float re, float im) {c->re=re;c->im=im;}  
  
Complex complexCreate(float re, float im) { Complex tmp=malloc(sizeof(struct complex));  
    init(tmp,re,im); }  
  
Complex complexAssignment(Complex dst, Complex src){  
    if (dst==src) return dst;  
    else {  
        void complexCopy(Complex dst, Complex src){ init(dst,src->re,src->im); }  
        void complexPrint(Complex c){ printf("%f + %fi\n", c->re, c->im); }  
        void complexDestroy(Complex c){ free(c); }  
        Complex tmp=malloc(sizeof(struct complex));  
        init(tmp,re,im);  
        complexCopy(dst,tmp);  
        free(tmp);  
        return dst;  
    }  
}
```

Must pass this parameter to know which instance the function applies to

Special functions for “assignment”. Why regular “=” does not work?

Since passed as pointer, the implementation might inadvertently modify it

# And in C++? complex.hpp

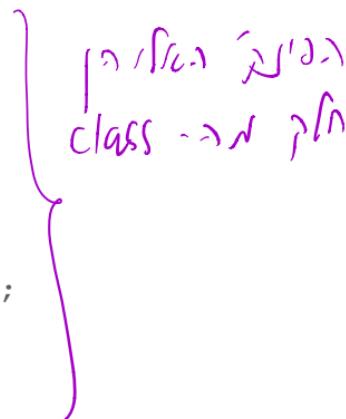
```
class Complex{
private:
    float re; float im;
    void init(float _re, float _im);

public:
    Complex();
    Complex(float _re, float _im);
    Complex(const Complex& other);

    ~Complex();

    Complex& operator=(const Complex& other);

    void print() const;
};
```



# And in C++? complex.cpp

```
#include "complex.hpp"

void Complex::init(float _re, float _im){re=_re;_im=im; }

Complex::Complex(){init(0.0,0.0);}

Complex::Complex(float _re, float _im){ init(_re,_im); }

Complex::Complex(const Complex& other){ init(other.re,other.im); }

Complex::~Complex(){}

Complex::Complex& operator=(const Complex& other){init(other.re,other.im); return
*this; }

void Complex::print() const { printf("%f %f\n",this->re,this->im); }
```

# Let's compare

```
class Complex{  
private:  
    float re; float im;  
    void init(float _re, float _im)  
  
public:  
    Complex();  
    Complex(float _re, float _im);  
    Complex(const Complex& other);  
  
    ~Complex();  
  
    Complex& operator=(const Complex&  
other);  
  
    void print() const;  
};
```

Declare the class with  
data and functions

Member  
variables

Member functions

Struct (in C) is only for  
data

```
struct complex{  
    float re; float im;  
}  
typedef struct complex * Complex;  
  
static void init(Complex c, float re,  
float im) {c->re=re;c->im=im;}
```

# Let's compare

```
class Complex{
private:
    float re; float im;
    void init(float _re, float _im);

public:
    Complex();
    Complex(float re, float im);
    Complex(const Complex & other);

    ~Complex();

    Complex& operator=(const Complex& other);

    void print() const;
};
```

```
struct complex{
    float re; float im;
}
typedef struct complex * Complex;

static void init(Complex c, float re,
float im) {c->re=re;c->im=im;}
```

not needed

not needed

Declare what can and cannot be seen by  
**instances of other classes:**  
**guarantees encapsulation**

# Let's compare

The function belongs to class  
Complex!

```
Complex::Complex(){init(0.0,0.0);}  
Complex::Complex(float _re, float  
_im){ init(_re, _im); }
```

Constructor: special function with  
**a reserved name**. Called when  
the instance of that class is  
*created*

```
Complex complexCreate(float re, float  
im) { Complex tmp=malloc(sizeof(struct  
complex));  
    complexInit(tmp, re, im); }
```

No need to allocate the object  
itself

# Let's compare

```
Complex::Complex(){init(0.0,0.0);}
Complex::Complex(float _re, float
_im){ init(_re,_im); }
```

We may have more than one  
constructor/function with the same  
name but different parameters.  
**(overloading)**

```
Complex complexCreate(float re, float
im) { Complex tmp=malloc(sizeof(struct
complex));
    complexInit(tmp,re,im); }
```

No need to allocate the object  
itself

# Let's compare

```
Complex::Complex(const Complex& other) {  
    init(other.re, other.im);  
}
```

```
Complex complexClone(Complex c) { return  
    complexCreate(c->re, c->im); }
```

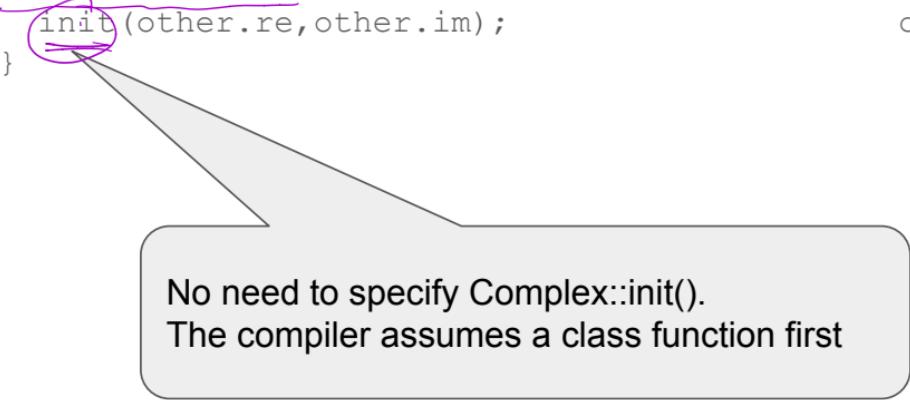


This is a special “copy” constructor called when creating a new instance **from the existing one**

# Let's compare

```
Complex::Complex(const Complex& other) {  
    init(other.re,other.im);  
}
```

```
Complex complexClone(Complex c) { return  
    complexCreate(c->re,c->im); }
```



No need to specify Complex::init().  
The compiler assumes a class function first

# Let's compare

member functions are invoked on a specific object implicitly!

```
void Complex::print() const {  
    printf("%f %f\n", this->re, this->im);  
}
```

This is a pointer to the current object on which the function is invoked.

Provided by C++

No need to pass this pointer

```
void complexPrint(Complex c) {  
    printf("%f %f\n", c->re, c->im);  
}
```

# Let's compare

This function **cannot** modify any of the class members! Part of the class interface.

```
void Complex::print() const {  
    printf("%f %f\n", this->re, this->im; }
```

```
void complexPrint(Complex c) {  
    printf("%f %f\n", c->re, c->im; }
```

# Let's compare

```
Complex::~Complex() {}
```



Destructor: special function called when the object is *destroyed*

```
void complexDestroy(Complex c) {  
    free(c); }
```



No need to deallocate memory of the object

# Let's compare

```
Complex::Complex& operator=(const  
Complex&  
other){init(other.re, other.im); return  
*this;}
```

Assignment operator. Allows to reimplement the copy logic

```
void complexCopy(Complex dst, Complex  
src){ init(dst,src->re,src->im); }
```

No need to define awkward special names for standard operations

# How do we use it?

```
#include "complex.hpp" // all the interfaces  
    (constructors and copy for multiloc (rs))  
int main(){  
    Complex* a=new Complex(1.0,0.0); // create instance on heap  
    Complex* b=new Complex(2.0,0.0); // another instance  
  
    *b=*a; // assignment  
    a->print(); // print object a  
    b->print(); // print object b  
  
    delete a; // invokes ~Complex() on a  
    delete b;  
    (free destroy j's object rs)  
    return 0;  
}
```

(Constructors and copy for multiloc (rs))

(C -> assignment)

(free destroy j's object rs)

# And now the same but allocated on stack

```
#include "Complex.hpp" // all the interfaces

int main(){
    Complex a(1.0,0.0); // create instance of Complex on stack
    Complex b(2.0,0.0); // another instance on stack

    b=a; // regular assignment, but invokes Complex::operator=
    a.print(); b.print();
    return 0;
}

} // the objects are destroyed automatically
```

# Let's delve into details

- References and Consts
- Constructors
- Destructors
- Assignment operators

# References

Syntax:

```
int x;  
int &y=x; // y and x are two names of the same variable  
y=3; // x is also 3;  
x=5; // y is also 5;
```

(*רִשְׁוֹת גַּם x, y  
הַלְּכָה אֶגְזָנָה  
פְּרָטָה לְהַ*)

Semantically equivalent to:

```
int x;  
int* y=&x;  
*y=3; // x is also 3;  
x=5; // *y is also 5;
```

# References are important in functions

Reminder: passing by reference or by value?

(C - 2)

```
void increment_1(int a){ a++; }  
void increment_2(int *a){ (*a)++; }
```

```
int main(){  
    int x=1;  
    increment_1(x); printf("%d\n",x); 1 ✓ by Val  
    increment_2(x); printf("%d\n",x); 2 by ref  
}
```

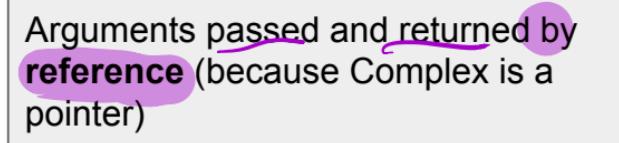
What is the output?

# References are important in functions

Passing arguments/return values by reference?

// C version:

```
Complex complexInc(Complex c){ c->re++; return c;}
```



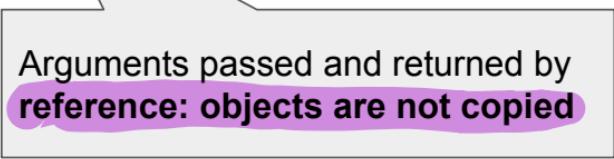
Arguments passed and returned by  
**reference** (because Complex is a  
pointer)

# References are important in functions

Passing arguments/return values by reference?

// C++ version:

```
Complex& complexInc(Complex& c){ c.increment(); return c;}
```



Arguments passed and returned by  
**reference: objects are not copied**

# Const

Const is an attribute that prohibits changes to the respective entity.

Const variables:

```
const int a=4;    // must be initialized where declared or in  
the constructor
```

```
a=6; // compilation error
```

Const references:

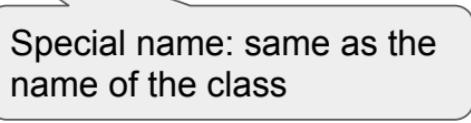
```
int foo(const Complex& c) { c.re=3; } // compilation error!
```

Const pointers - the memory they point to is const, not the pointer!

```
const char* v="string";  
v[2] = "z"; // compilation error!!  
v = "another string"; // OK
```

# Constructors

```
class Complex{  
  
public:  
    Complex(); // default constructor  
    Complex(float _re, float _im); // two arguments  
    Complex(const Complex& other); // copy-constructor  
};
```



Special name: same as the  
name of the class

# (constructor) Using a stack from file input output

## Constructors

**Constructors are invoked when an object is created**

```
Complex c; // on stack. default constructor is called
```

```
Complex c(1.0,0.0); // on stack. Constructor with parameters is called
```

```
Complex dst(src); // on stack. Copy of src. copy-constructor
```

```
Complex dst=src; // on stack. Copy of src.copy-constructor
```

```
Complex* ptr_c=new Complex(); // on heap, no-arg constructor
```

```
Complex* ptr_c=new Complex(1.0,0.0); // on heap
```

```
Complex ptr_dst=new Complex(*src); // on heap, copy-constructor
```

Copy constructors are called automatically when  
passing by value!

```
int foo(Complex c) { ...};  
foo(x); // copy of x. copy-constructor
```

```
Complex bar(){ Complex tmp; // default constructor  
return tmp; } //copy of tmp, copy-constructor
```

By Val

# How to avoid unnecessary constructor invocation?

Pass by reference!

```
int foo(Complex &c) { ...};  
foo(x); // no copy-constructor invoked
```

```
//BUG!!!  
Complex& bar() {  
    Complex tmp; // default constructor  
    return tmp; // tmp itself is passed back, but it's on stack  
}
```

and pass by value instead  
return type for copy const

# What if no constructors are defined?

C++ compiler **implicitly** defines

default constructor if no other non-copy constructors are declared

copy constructor if **no copy-constructor** are declared

Implicit default constructor does nothing (trivial)

Implicit copy constructor copies bit-by-bit (like struct passed by value)

**Works correctly for simple cases**

(*Yufei says it's ok*)

# Will implicit default constructor work?

```
class MyString{  
private:  
    int len;  
    char* str;  
public:  
    MyString(int _len, const char* _str){  
        len=_len; str=new char[len];  
        strncpy(str,_str,len);  
    }  
    bool isEmpty() const{return _len==0;}  
};
```

Does it

# Will implicit default constructor work?

```
class MyString{  
private:  
    int len;  
    char* str;  
public:  
    MyString(int _len, const char* _str){  
        len=_len; str=new char[len];  
        strncpy(str,_str,len);  
    }  
    bool isEmpty() const{return _len==0;}  
};
```

No. A non-default constructor defined.

# Will default copy constructor work?

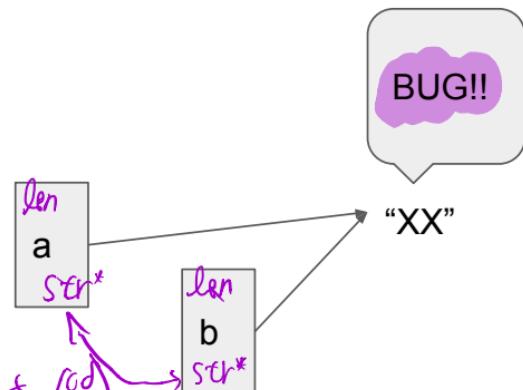
```
class MyString{
private:
    int len;
    char* str;
public:
    MyString(int _len, const char* _str){
        len=_len; str=new char[len];
        strncpy(str,_str,len);
    }
    bool isEmpty() const{return _len==0;}
};

int main(){
    MyString a(2,"XX");
    MyString b=a;
```

# Will default copy constructor work?

```
class MyString{  
private:  
    int len;  
    char* str;  
public:  
    MyString(int _len, const char* _str){  
        len=_len; str=new char[len];  
        strncpy(str,_str,len);  
    }  
    bool isEmpty() const{return _len==0;}  
};  
  
int main(){  
    MyString a(2,"XX");  
    MyString b=a;
```

(result b->str points to the same memory as a->str)



# a & the constructor

## Constructors: new (correct) syntax

```
class A{  
    private:  
        const int first=0;  
        float second; = 1.0  
        double third; = 2.0  
    };  
    A():second(1.0), third(2.0){}  
};
```

Inline initialization allowed

Complex C++ syntax is evil

A(): ..., C(1.0, 1.0){ }

Explicit initialization with *order*

# Destructors

Destructor has no arguments

Invoked right before the object is deleted

```
class Ex{};  
int main() {  
    Ex a;  
    Ex* b=new Ex();  
    delete b;  
}
```

Destructor b is  
invoked here

Destructor a is  
invoked here

# Implicit destructor is empty!

Will implicit destructor work for this class?

NO

```
class MyString{  
private:  
    int len;  
    char* str;  
public:  
    MyString(int _len, const char* _str){  
        len=_len; str=new char[len];  
        strncpy(str,_str,len);  
    }  
    bool isEmpty() const{return _len==0;}  
};
```

( This program prints '123' because the implicit destructor is empty )

# Implicit destructor is empty!

Will implicit destructor work for this class?

```
class MyString{  
private:  
    int len;  
    char* str;  
public:  
    MyString(int _len, const char*  
             len=_len;  str=new char[len];  
             strncpy(str,_str,len);  
    }  
    bool isEmpty() const{return _len==0;}  
  
~MyString(){  
    delete str;  
}  
};
```

No, it causes  
memory leak!

# Const functions

```
class Ex{  
    int val;  
    void print() const{  
        cout<<val<<endl;  
    }  
};
```

This function cannot modify class members

The code defines a class named Ex with an integer member variable val. It contains a print function that outputs the value of val followed by a new line. The print function is declared as const, which is highlighted with a pink bracket. A callout arrow points from this const keyword to a text box containing the explanatory text: "This function cannot modify class members".

**const** functions allow more precise definition of the class interface

# Assignment operator

```
class Complex{  
private:  
    float re; float im;  
    void init(float _re, float _im);  
  
public:  
...  
    Complex& operator=(const Complex& other);  
...  
};
```

```
int main(){  
    Complex a,b;
```

```
a=b;
```

copy con. fn wch? w/?

The signature is always of the form:  
T& operator=(const T&)

Why does it return T&??

( $a = b = c$  ref. to 'r')

# Assignment operator vs copy constructor

```
class Complex{  
...  
public:  
    Complex(const Complex& other);  
    Complex& operator=(const Complex& other);  
};  
  
int main() {  
    Complex a,b,c;  
  
    a=b=c; // operator=, chaining is enabled because operator= returns Complex&  
    Complex d=c; // copy constructor. Object d is created. c is used for initialization  
}  
        (Copy con.)
```

# Implicit operator=

Similarly to copy constructor, if `operator=` is not defined by the user, its default is bit-by-bit copy

Will implicit operator= work for this class?

(Copy con. & J/Kr. wo Aktion) ✓  
1.5

```
class MyInt{
private:
    int* val;
public:
    MyInt(int _val):val(new int(_val)) {}
    void inc() { *val++; }
}
```

# Implicit operator=

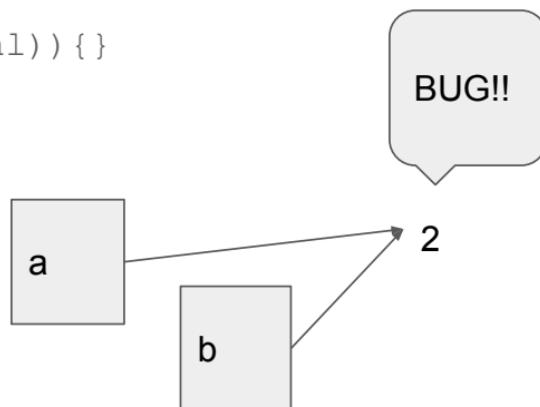
Similarly to copy constructor, if operator= is not defined by the user, its default is bit-by-bit copy

Will implicit operator= work for this class?

```
class MyInt{  
private:  
    int* val;  
public:  
    MyInt(int _val):val(new int(_val)) {}  
    void inc() { *val++; }  
}  
  
int main(){  
    MyInt a(1), b(2);  
    a=b;  
    a.inc();
```

This changes b too

BUG!!



# Implementing operator=(). Try 1

```
class MyInt{  
private:  
    int* val=null;  
public:  
    MyInt() {}  
  
    MyInt(int _val):val(new int(_val)) {}  
  
    void inc(){ *val++; } (null pointer fix)  
  
    MyInt& operator=(const MyInt& other) {  
        val=new int(other); copy  
        return *this; delete from  
    }  
}
```

# Implementing operator=().

```
class MyInt{
private:
    int* val=null;
public:
    MyInt() {}

    MyInt(int _val):val(new int(_val)) {}

    void inc(){ *val++; }

    MyInt& operator=(const MyInt& other) {
        delete val;
        val=new int(other);
        return *this;
    }
}
```

# 6.088 Intro to C/C++

Day 5: Inheritance & Polymorphism

Eunsuk Kang & Jean Yang

# In the last lecture...

Objects: Characteristics & responsibilities

Declaring and defining classes in C++

Fields, methods, constructors, destructors

Creating & deleting objects on stack/heap

Representation invariant

# Today's topics

Inheritance

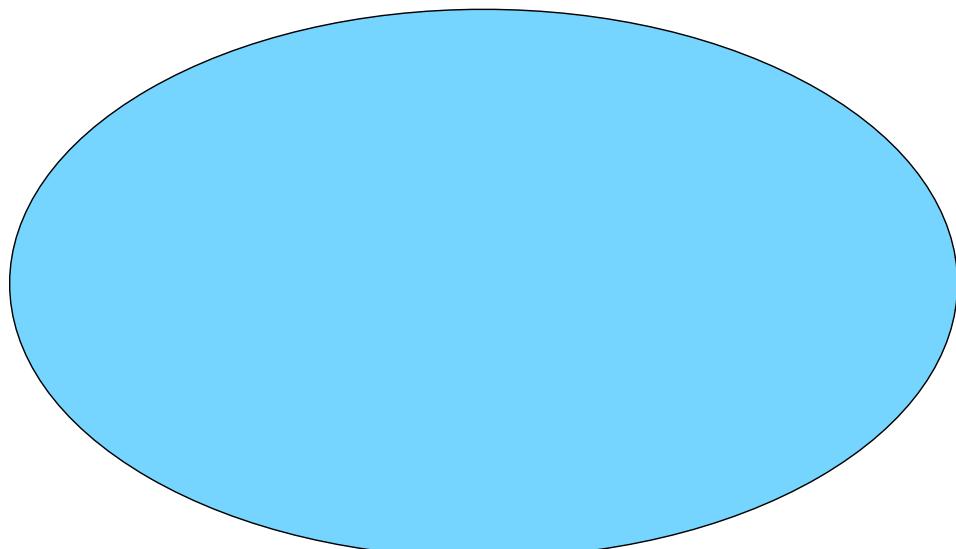
Polymorphism

Abstract base classes

# Inheritance

# Types

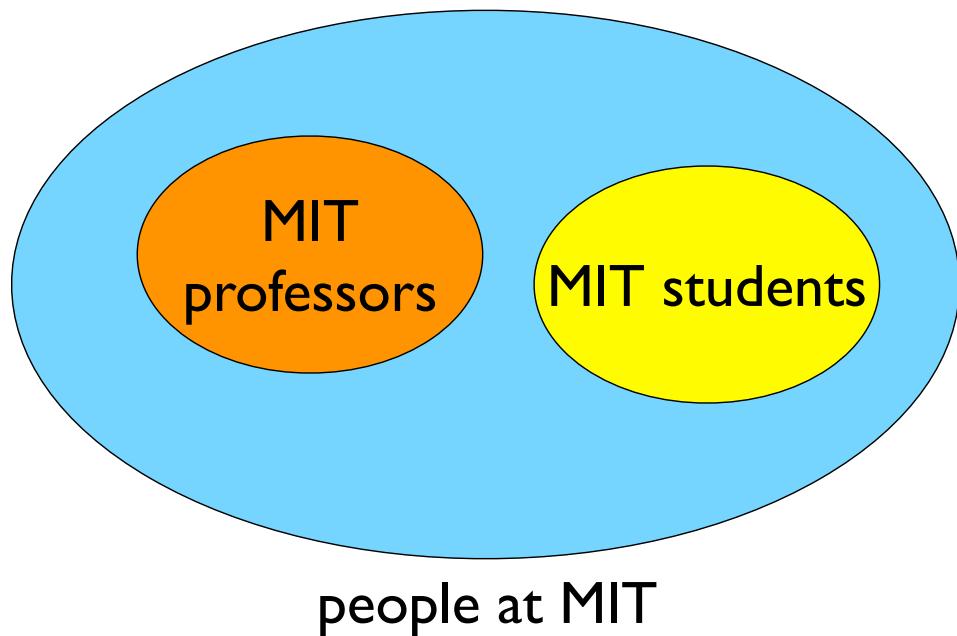
A class defines a set of objects, or a **type**



people at MIT

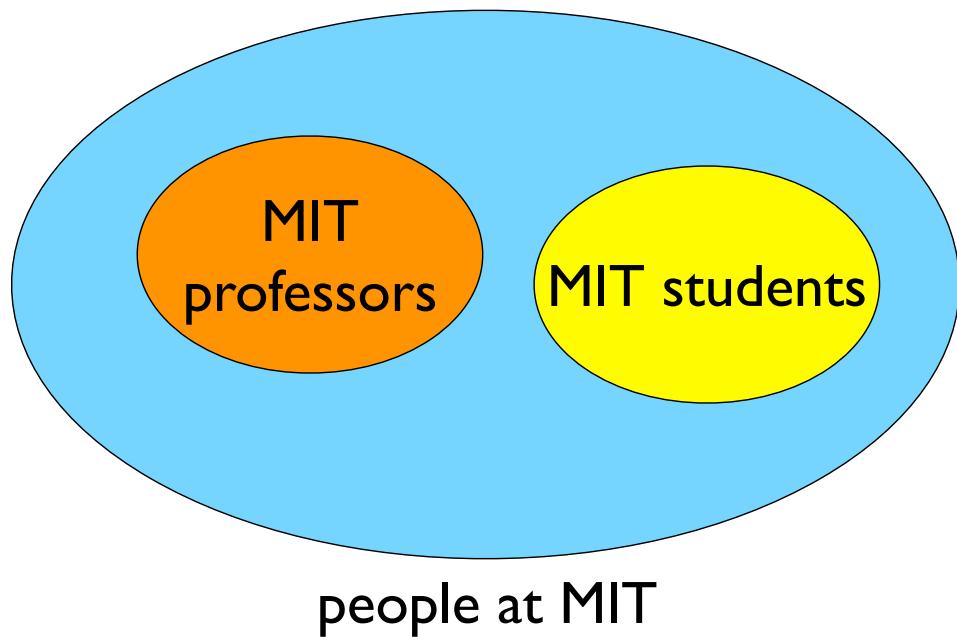
# Types within a type

Some objects are distinct from others in some ways

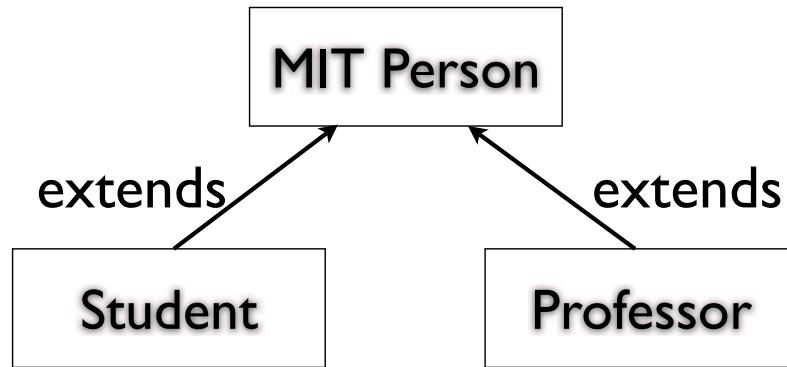


# Subtype

MIT professor and student are **subtypes** of MIT people

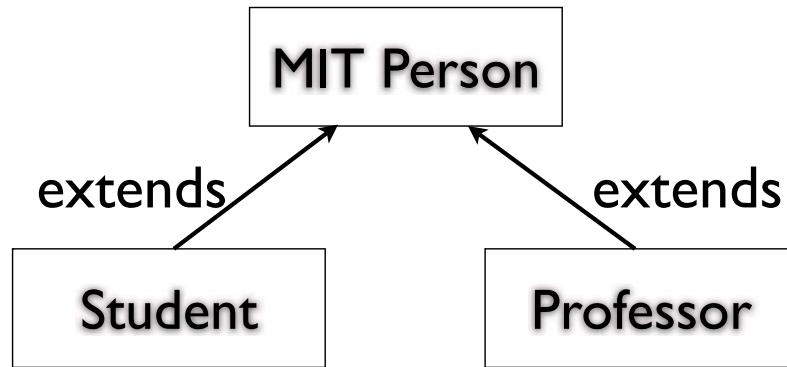


# Type hierarchy



What characteristics/behaviors do people at MIT have in common?

# Type hierarchy



What characteristics/behaviors do people at MIT have in common?

- ▶ name, ID, address *(n·d·env)*
- ▶ change address, display profile *(n·c)*

C ->

(ADT)  $\rightarrow$  struct ST {

(ADT (ADT))  $\rightarrow$  struct MIT\_P {

-> ID... ;

}

add cours();

{

MIT Person

extends

Student

extends

Professor

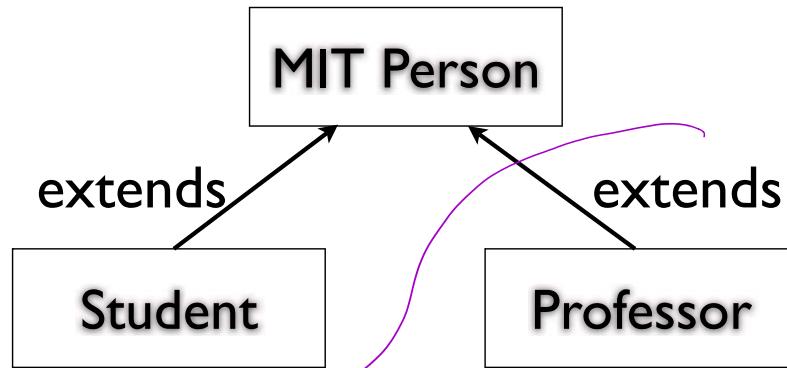
מבחן מילויים  
בנין MIT\_P  
בנין MIT\_P  
בנין MIT\_P  
בנין MIT\_P  
בנין MIT\_P

## Type hierarchy

What things are special about students?

- ▶ course number, classes taken, year
- ▶ add a class taken, change course

# Type hierarchy



What things are special about professors?

- ▶ course number, classes taught, rank (assistant, etc.)
- ▶ add a class taught, promote

# Inheritance

A subtype **inherits** characteristics and behaviors of its base type.

e.g. Each MIT student has

Characteristics:

name

ID

address

course number

classes taken

year

Behaviors:

display profile

change address

add a class taken

change course

# Base type: MITPerson

```
#include <string>

class MITPerson {

protected:
    int id;
    std::string name;
    std::string address;

public:

    MITPerson(int id, std::string name, std::string address);

    void displayProfile();
    void changeAddress(std::string newAddress);

};
```

# Base type: MITPerson

```
#include <string>

class MITPerson {
protected:
    int id;
    std::string name;
    std::string address;

public:
    MITPerson(int id, std::string name, std::string address);
    void displayProfile();
    void changeAddress(std::string newAddress);
};
```

**namespace prefix**

The diagram shows a black arrow originating from the text "namespace prefix" and pointing towards the "std::string name;" line in the code. The word "name" is enclosed in an oval.

# Base type: MITPerson

```
#include <string>

class MITPerson {
protected:           access control
    int id;
    std::string name;
    std::string address;

public:
    MITPerson(int id, std::string name, std::string address);
    void displayProfile();
    void changeAddress(std::string newAddress);

};
```

# Access control

**Public**

accessible by anyone

**Protected**

accessible inside the class and by all of its subclasses

**Private**

accessible only inside the class, NOT including its  
subclasses

# Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

    int course;
    int year;      // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);

};
```

# Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

    int course;
    int year;      // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);

};
```

dynamic array,  
part of C++ standard library

# Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

    int course;
    int year;      // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);

};
```

# Constructing an object of subclass

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

    int course;
    int year;      // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);

};
```

# Constructing an object of subclass

```
// in Student.cc
Student::Student(int id, std::string name, std::string address,
                  int course, int year) : MITPerson(id, name, address)
{
    this->course = course;
    this->year = year;
}
```

```
// in MITPerson.cc
MITPerson::MITPerson(int id, std::string name, std::string address){
    this->id = id;
    this->name = name;
    this->address = address;
}
```

# Calling constructor of base class

call to the base constructor

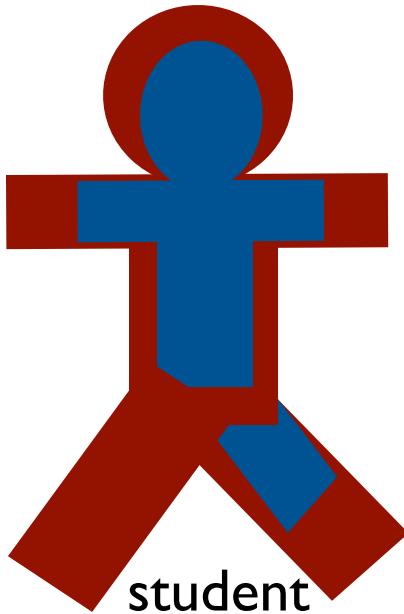
```
// in Student.cc
Student::Student(int id, std::string name, std::string address,
                  int course, int year) : MITPerson(id, name, address)
{
    this->course = course;
    this->year = year;
}
```



```
// in MITPerson.cc
MITPerson::MITPerson(int id, std::string name, std::string address){
    this->id = id;
    this->name = name;
    this->address = address;
}
```

# Constructing an object of subclass

```
Student* james =  
    new Student(971232, "James Lee", "32 Vassar St.", 6, 2);
```



name = "James Lee"  
ID = 971232  
address = "32 Vassar St."  
course number = 6  
classes taken = none yet  
year = 2

# Overriding a method in base class

```
class MITPerson {  
protected:  
    int id;  
    std::string name;  
    std::string address;  
public:  
    MITPerson(int id, std::string name, std::string address);  
    void displayProfile();  
    void changeAddress(std::string newAddress);  
};
```

```
class Student : public MITPerson {  
int course;  
int year;      // 1 = freshman, 2 = sophomore, etc.  
std::vector<Class*> classesTaken;  
public:  
    Student(int id, std::string name, std::string address,  
            int course, int year);  
    void displayProfile(); // override the method to display course & classes  
    void addClassTaken(Class* newClass);  
    void changeCourse(int newCourse);  
};
```

# Overriding a method in base class

```
void MITPerson::displayProfile() { // definition in MITPerson
    std::cout << "-----\n";
    std::cout << "Name: " << name << " ID: " << id
        << " Address: " << address << "\n";
    std::cout << "-----\n";
}
```

```
void Student::displayProfile(){ // definition in Student
    std::cout << "-----\n";
    std::cout << "Name: " << name << " ID: " << id
        << " Address: " << address << "\n";
    std::cout << "Course: " << course << "\n";
    std::vector<Class*>::iterator it;
    std::cout << "Classes taken:\n";
    for (it = classesTaken.begin(); it != classesTaken.end(); it++){
        Class* c = *it;
        std::cout << c->getName() << "\n";
    }
    std::cout << "-----\n";
}
```

# Overriding a method in base class

```
MITPerson* john =
    new MITPerson(901289, "John Doe", "500 Massachusetts Ave.");
Student* james =
    new Student(971232, "James Lee", "32 Vassar St.", 6, 2);
Class* c1 = new Class("6.088");
james->addClassTaken(c1);
john->displayProfile();
james->displayProfile();
```

-----  
Name: John Doe ID: 901289 Address: 500 Massachusetts Ave.

-----  
-----  
Name: James Lee ID: 971232 Address: 32 Vassar St.

Course: 6

Classes taken:

6.088

# Polymorphism

# Polymorphism

Ability of type A to appear as and be used like another type B

e.g. A Student object can be used in place of an  
MITPerson object

# Actual type vs. declared type

Every variable has a **declared type** at compile-time

But during runtime, the variable may refer to an object with an **actual type**  
(either the same or a subclass of the declared type)

```
MITPerson* john =  
    new MITPerson(901289, "John Doe", "500 Massachusetts Ave.");  
MITPerson* steve =  
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
```

What are the declare types of john and steve?  
What about actual types?

# Calling an overridden function

```
MITPerson* steve =  
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);  
  
steve->displayProfile();
```

# Calling an overridden function

```
MITPerson* steve =  
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);  
  
steve->displayProfile();
```

```
-----  
Name: Steve ID: 911923 Address: 99 Cambridge St.  
-----
```

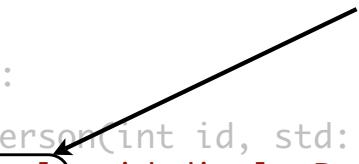
Why doesn't it display the course number and classes taken?

# Virtual functions

Declare overridden methods as **virtual** in the base

```
class MITPerson {  
  
protected:  
    int id;  
    std::string name;  
    std::string address;  
  
public:  
    MITPerson(int id, std::string name, std::string address);  
    virtual void displayProfile();  
    virtual void changeAddress(std::string newAddress);  
};
```

‘virtual’ keyword



What happens in other languages (Java, Python, etc.)?

# Calling a virtual function

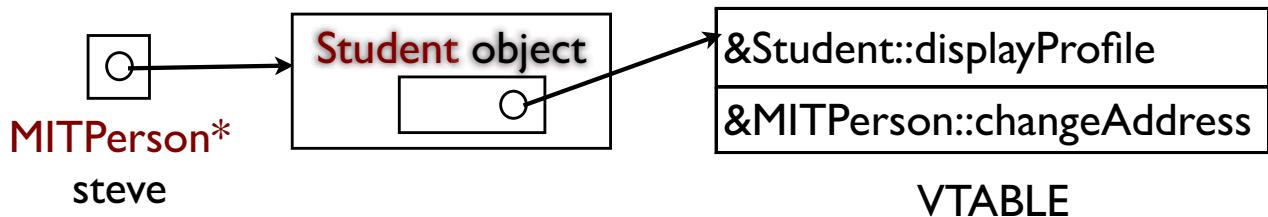
```
MITPerson* steve =  
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);  
  
steve->displayProfile();
```

```
-----  
Name: Steve ID: 911923 Address: 99 Cambridge St.  
Course: 18  
Classes taken:  
-----
```

# What goes on under the hood?

## Virtual table

- ▶ stores pointers to all virtual functions
- ▶ created per each class
- ▶ lookup during the function call



Note “`changeAddress`” is declared `virtual` in but not overridden

# Virtual destructor

Should destructors in a base class be declared as virtual? Why or why not?

# Virtual destructor

Should destructors in a base class be declared as virtual? Why or why not?

**Yes!** We must always clean up the mess created in the subclass (otherwise, risks for memory leaks!)

# Virtual destructor example

```
class Base1 {
public:
    ~Base1() { std::cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
    ~Derived1() { std::cout << "~Derived1()\n"; }
};

class Base2 {
public:
    virtual ~Base2() { std::cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
    ~Derived2() { std::cout << "~Derived2()\n"; }
};

int main() {
    Base1* bp = new Derived1; // Upcast
    delete bp;
    Base2* b2p = new Derived2; // Upcast
    delete b2p;
}
```

# Virtual destructor in MITPerson

```
class MITPerson {  
  
protected:  
    int id;  
    std::string name;  
    std::string address;  
  
public:  
  
    MITPerson(int id, std::string name, std::string address);  
    ~MITPerson();  
    virtual void displayProfile();  
    virtual void changeAddress(std::string newAddress);  
};  
  
MITPerson::~MITPerson() { }
```

# Virtual constructor

Can we declare a constructor as virtual? Why or why not?

# Virtual constructor

Can we declare a constructor as virtual? Why or why not?

No, not in C++. To create an object, you must know its exact type. The VPTR has not even been initialized at this point.

# Type casting

```
MITPerson* steve =  
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);  
Class* c1 = new Class("6.088");  
  
steve->addClassTaken(c1);
```

What will happen?

# Type casting

```
MITPerson* steve =  
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);  
Class* c1 = new Class("6.088");  
  
steve->addClassTaken(c1); X
```

Can only invoke methods of the declared type!

“addClassTaken” is not a member of MITPerson

# Type casting

```
MITPerson* steve =
    new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
Class* c1 = new Class("6.088");

Student* steve2 =
    dynamic_cast<Student*>(steve);

steve2->addClassTaken(c1); // OK
```

Use “`dynamic_cast<...>`” to **downcast** the pointer

# Static vs. dynamic casting

Can also use “`static_cast<...>`”

```
Student* steve2 =  
    static_cast<Student*>(steve);
```

Cheaper but dangerous! No runtime check!

```
MITPerson* p = MITPerson(...);  
Student* s1 = static_cast<Student*>(p); // s1 is not checked! Bad!  
Student* s2 = dynamic_cast<Student*>(p); // s2 is set to NULL
```

Use “`static_cast<...>`” only if you know what you are doing!

# Abstract base class

# Abstract methods

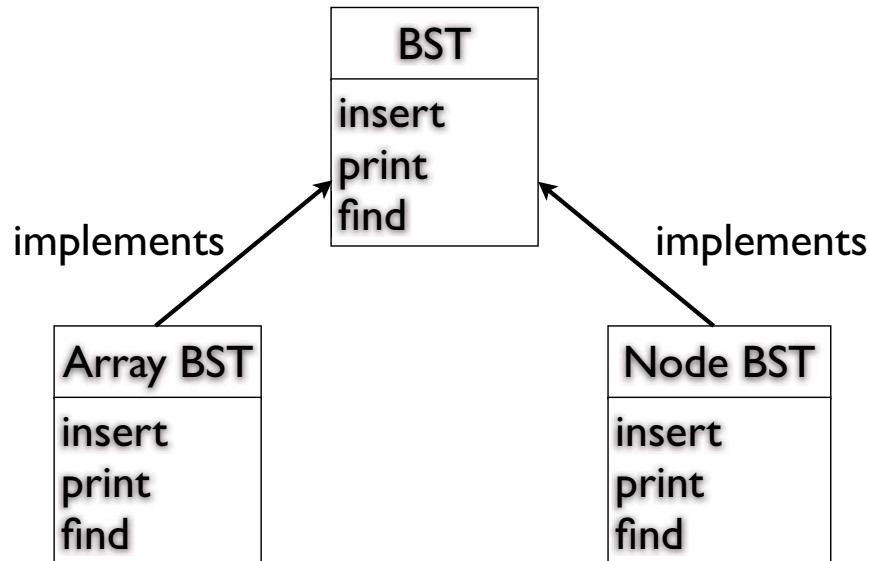
Sometimes you want to inherit only declarations, not definitions

A method without an implementation is called an **abstract method**

Abstract methods are often used to create an **interface**

# Example: Binary search tree

Can provide multiple implementations to BST



Decouples the client from the implementations

# Defining abstract methods in C++

Use **pure virtual functions**

```
class BST {  
  
public:  
    virtual ~BST() = 0;  
  
    virtual void insert(int val) = 0;  
    virtual bool find(int val) = 0;  
    virtual void print_inorder() = 0;  
};
```

(How would you do this in Java?)

# Defining abstract methods in C++

Use **pure virtual functions**

```
class BST {  
public:  
    virtual ~BST() = 0;  
  
    virtual void insert(int val) = 0;  
    virtual bool find(int val) = 0;  
    virtual void print_inorder() = 0;  
};
```

this says that “find” is **pure**  
(i.e. no implementation)

this says that “find” is **virtual**

# Defining abstract methods in C++

Can we have non-virtual pure functions?

```
class BST {  
  
public:  
    virtual ~BST() = 0;  
  
    virtual void insert(int val) = 0;  
    virtual bool find(int val) = 0;  
    virtual void print_inorder() = 0;  
};
```

# Abstract classes in C++

## Abstract base class

- ▶ a class with one or more pure virtual functions
- ▶ cannot be instantiated

```
BST bst = new BST(); // can't do this!
```

- ▶ its subclass must implement all of the pure virtual functions (or itself become an abstract class)

# Extending an abstract base class

```
class NodeBST : public BST {  
    Node* root;  
  
public:  
    NodeBST();  
    ~NodeBST();  
    void insert(int val);  
    bool find(int val);  
    void print_inorder();  
};  
// implementation of the insert method using nodes  
void NodeBST::insert(int val) {  
    if (root == NULL) {  
        root = new Node(val);  
    } else {  
        ...  
    }  
}
```

# Constructors in abstract classes

Does it make sense to define a constructor?  
The class will never be instantiated!

# Constructors in abstract classes

Does it make sense to define a constructor?  
The class will never be instantiated!

**Yes!** You should still create a constructor to initialize its members, since they will be inherited by its subclass.

# Destructors in abstract classes

Does it make sense to define a destructor?

The class will never be created in the first place.

# Destructors in abstract classes

Does it make sense to define a destructor?

The class will never be created in the first place.

**Yes!** Always define a **virtual** destructor in the base class, to make sure that the destructor of its subclass is called!

# Pure virtual destructor

Can also define a destructor as **pure**.

```
class BST {  
  
public:  
    virtual ~BST() = 0;  
  
    virtual void insert(int val) = 0;  
    virtual bool find(int val) = 0;  
    virtual void print_inorder() = 0;  
};
```

But must also provide a function body. Why?

```
BST::~BST() {}
```

# Until next time...

Homework #5 (due 11:59 PM Tuesday)

- ▶ Designing & implementing type hierarchy for simple arithmetic expressions

Next lecture

- ▶ templates
- ▶ common C++ pitfalls
- ▶ C/C++ interview questions & tricks

# References

Thinking in C++ (B. Eckel) **Free online edition!**

Essential C++ (S. Lippman)

Effective C++ (S. Meyers)

C++ Programming Language (B. Stroustrup)

Design Patterns (Gamma, Helm, Johnson, Vlissides)

Object-Oriented Analysis and Design with Applications (G. Booch, et. al)

# Extra slides

# Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"      what if this is private?
#include "Class.h"

class Student : public MITPerson {
    int course;
    int year;      // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);

};
```

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.088 Introduction to C Memory Management and C++ Object-Oriented Programming

January IAP 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.

# C++ Templates and STL

044101

# Generic data structures

Problem: create an ADT that can hold any data type

```
typedef void (* print_element_func) (void*);  
struct queue{  
    void* data;  
    size_t total;  
    size_t el_size;  
    print_element_func print_element;  
}  
void print_queue(struct queue* q) {  
    for (void* ptr=q->data, int i=0; i<total;  
        ptr+=((void*)((char*)ptr)+el_size), i++) {  
        q->print_element_func(ptr);  
    }  
}  
void print_element_string(void* e) { char* tmp=(char*)e; printf(tmp) }
```

# In C++ there is a special construct to do that

```
template<typename T>
class Queue{
    T* data;
    size_t total_el;
public:
.....
    void print(){
        for (int i=0;i<total;i++)    data[i].print();
    }
};

class String{
    const char* data;
    public: print(){ cout<< data << endl;
.....
}

int main(){
    Queue<String> queue_of_strings;
.....
    queue_of_strings.print();
```

# In C++ there is a special construct to do that

```
template<typename T>
class Queue{
    T* data;
    size_t total_el;
public:
    void print() {
        for (int i=0;i<total;i++)    data[i].print();
    }
};

int main() {
    Queue<String> queue_of_strings;
    .....
    queue_of_strings.print();
```

The **compiler (actually, preprocessor)** replaces T with real class

It knows what class to replace with at the time of **declaration**

# Templates are instantiated only when declared!

```
template<typename T>
class Queue{
    T* data;
    size_t total_el;
public:
    void print() {
        for (int i=0;i<total;i++)    data[i].print();
    }
};

int main() {
    Queue<String> queue_of_strings;

    Queue<char> queue_of_chars;
```

Compiler actually writes code with the class  
of this type

Compilation error! no char.print() defined!

# More complex example: Stack

```
template <typename T>
class Stack {
private:
    vector<T> elems;      // container from Standard Template Library (STL)

public:
    void push(T const&);  // push element
    void pop();           // pop element
    T top() const;        // return top element

    bool empty() const {   // return true if empty.
        return elems.empty();
    }
};
```

# Out-of-class function definition

```
template <typename T>
void Stack<T>::push (T const& elem) {
    // append copy of passed element
    elems.push_back(elem);
}
```

```
template <typename T>
void Stack<T>::pop () {
    if (elems.empty()) return;

    // remove last element
    elems.pop_back();
}
```

# Generated class (compile time) vs. object of the class (runtime) ?

```
int main() {  
  
    Stack<int>      intStack; // stack of ints  
  
    Stack<int>      anotherIntStack; // stack of ints  
  
    Stack<double>* doubleStack; // stack of doubles  
  
    Stack<Stack<double> *> complexStack; // stack of  
    pointers to stacks of double  
  
}
```

| Compile time | Run time |
|--------------|----------|
| yes          | yes      |
| no           | yes      |
| yes          | no       |
| yes          | yes      |

# Function templates

```
template <typename T>
const T& Max (const T& a, const T& b) {
    return a < b ? b:a;
}

int main () {
    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl; // compiler generates a
new function for INT

    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl; // compiler generates
a new function for DOUBLE
.....
}
```

# Standard Template Library

STL is a standard C++ library with containers, algorithms and iterators

**Containers:** used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.

**Algorithms:** act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

**Iterators:** used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

# Vector: auto-resizable array

```
#include <iostream>
#include <vector>

using namespace std;

int main() {

    // create a vector to store int
    vector<int> vec;
    int i;

    // display the original size of vec
    cout << "vector size = " << vec.size() << endl;

    // push 5 values into the vector
    for(i = 0; i < 5; i++) {
        vec.push_back(i);
    }
}
```

# Accessing elements

```
// access 5 values from the vector
for(i = 0; i < 5; i++) {
    cout << "value of vec [ " << i << "] = " <<vec[i] << endl;
}

// use iterator to access the values
vector<int>::iterator v = vec.begin();

while( v != vec.end()) {
    cout << "value of v = " << *v << endl;
    v++;
}
```

Simple access to  
elements via [ ]



# Using iterators

```
// access 5 values from the vector  
for(i = 0; i < 5; i++) {  
    cout << "value of vec [ " << i << " ] = " <<vec[i] << endl;  
}
```

Get the iterator  
to the beginning

```
// use iterator to access the values  
vector<int>::iterator v = vec.begin(),
```

```
while( v != vec.end()) {  
    cout << "value of v = " << *v << endl,  
    v++;
```

Use iterator to advance to  
the next element

Get the iterator  
to the beginning

Many iterators exist:  
random-access,  
sequential, read-only, insert, etc.

# How can we use STL algorithms?

Given std::vector v, find out how many 4 it contains?

```
int fours = 0;  
for (int i = 0; i < v.size(); ++i) {  
    if (v[i] == 4) fours++;  
}
```

# And we can use the `stl::count` function

```
int fours=stl::count (v.begin(), v.end(), 4);
```

# Another interesting example: hashtable

```
#include <iostream>
#include <map>
#include <string>
#include <iterator>

int main()
{
    std::map<std::string, int> mapOfWords;
// Inserting data in std::map
    mapOfWords.insert(std::make_pair("earth", 1));
    mapOfWords.insert(std::make_pair("moon", 2));
    mapOfWords["sun"] = 3;
// Will replace the value of already added key i.e. earth
    mapOfWords["earth"] = 4;
```

# Hashtable cont.

```
// Iterate through all elements in std::map
std::map<std::string, int>::iterator it = mapOfWords.begin();
while(it != mapOfWords.end())
{
    std::cout<<it->first<<" :: "<<it->second<<std::endl;
    it++;
}
// Check if insertion is successful or not
if(mapOfWords.insert(std::make_pair("earth", 1)).second == false)
{
    std::cout<<"Element with key 'earth' not inserted because already existed"<<std::endl;
}
// Searching element in std::map by key.
if(mapOfWords.find("sun") != mapOfWords.end())
    std::cout<<"word 'sun' found"<<std::endl;
if(mapOfWords.find("mars") == mapOfWords.end())
    std::cout<<"word 'mars' not found"<<std::endl;
return 0;
}
```

# Summary

1. Templates are extremely useful tools for generic ADTs and beyond
2. Compile-time mechanism: all classes are generated by the compiler for each type
3. Standard Template Library includes a lot of very useful containers (list, vector, deque, set, map, bitset), useful algorithms and much more..

Most of the data structures for future courses you will find in STL.

# Course recap

## Goals

- Understand how **computers actually work** from the software perspective
- Hands on: learn and practice **Systems Programming** basics
- Useful outcomes:
  - Understand hardware and software interact
  - Become more effective programmer
  - Prepare for later courses: OS, Compilers, Networking, Computer Structure, Computer Security

Your **first zipper course** in Computer Engineering

Representation  
Assembly  
Compiler  
Linker  
Binary and ELF  
Buffer overflow

Bash  
Makefile  
Files  
vi, git, valgrind, gdb

Advanced C and ADT  
Intro to C++

# Course recap

## Goals

- Understand how **computers actually work** from the software perspective
- Hands on: learn and practice **Systems Programming** basics
- Useful outcomes:
  - Understand hardware and software interact
  - Become more effective programmer
  - Prepare for later courses: OS, Compilers, Networking, Computer Structure, Computer Security

Your **first zipper course** in Computer Engineering

Representation  
Assembly  
Compiler  
Linker  
Binary and ELF  
Buffer overflow

Bash  
Makefile  
Files  
vi, git, valgrind, gdb

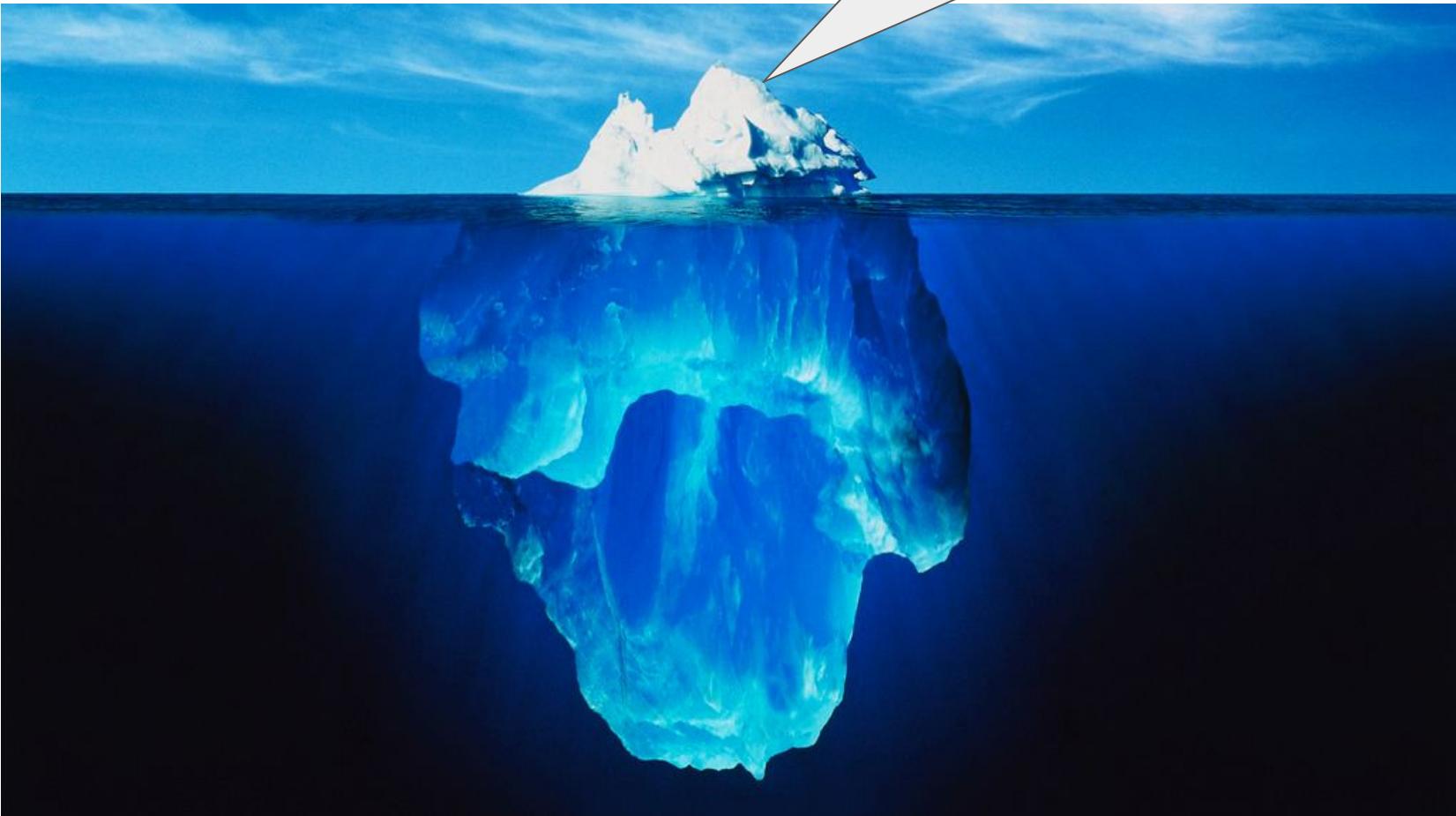
Advanced C and ADT  
Intro to C++



So what do you know now?

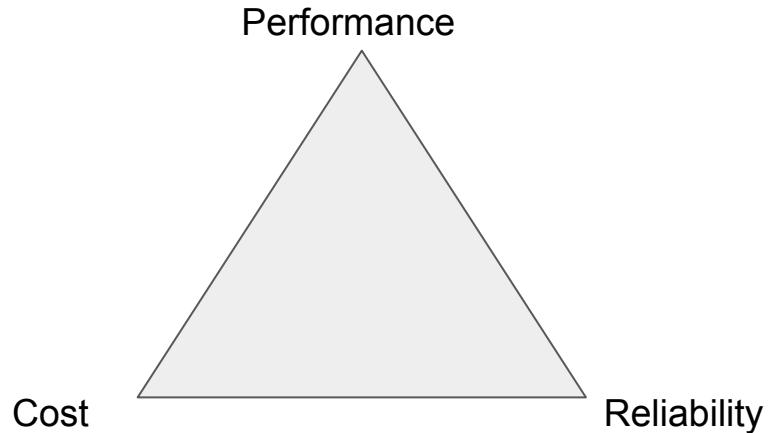
# So what do you know now?

You are here...

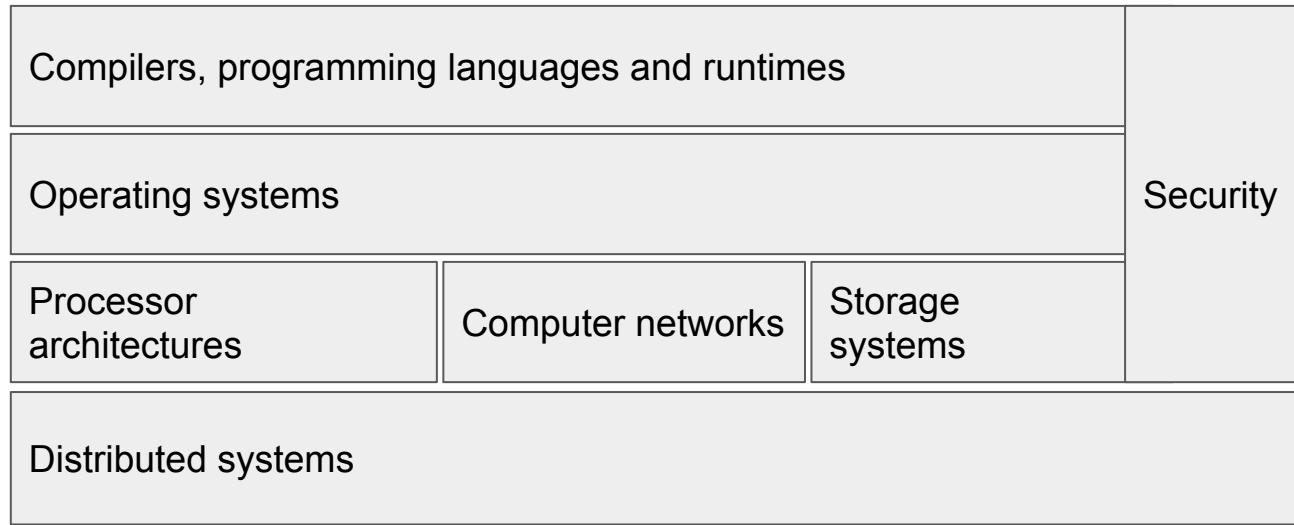


# Let's talk about computer systems

Engineering is the art of tradeoffs



# Computer systems are built of multiple components

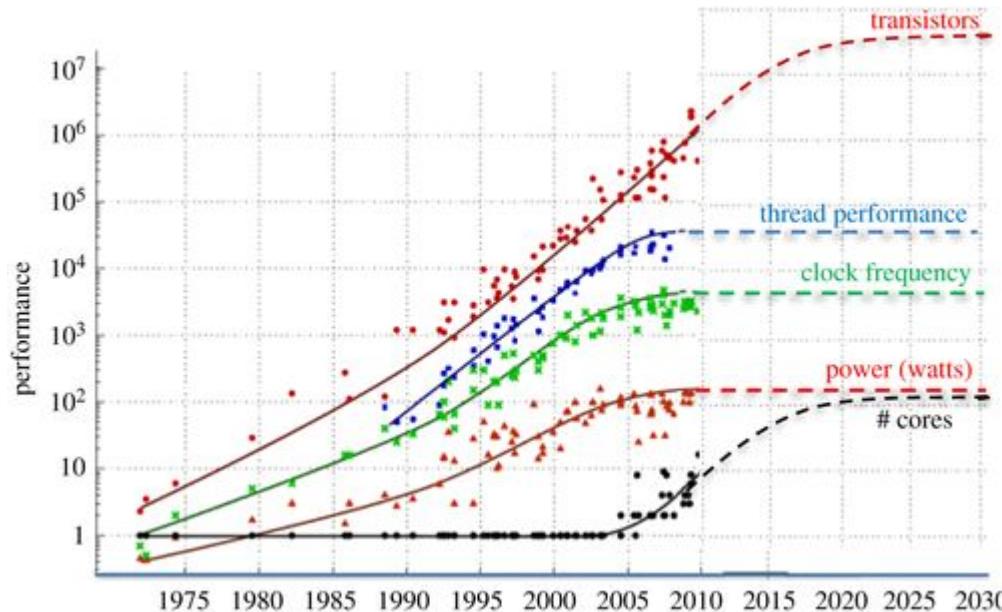


These components interact in a lot of ways.

We must have an in-depth understanding of the mechanisms to build a functioning system

# Interesting questions...

Processor architectures: we can't make faster CPUs



# How do we make progress?

Use **accelerators**! 10-10000x faster

Specialized units to perform specific tasks

New highly *parallel* computing architectures

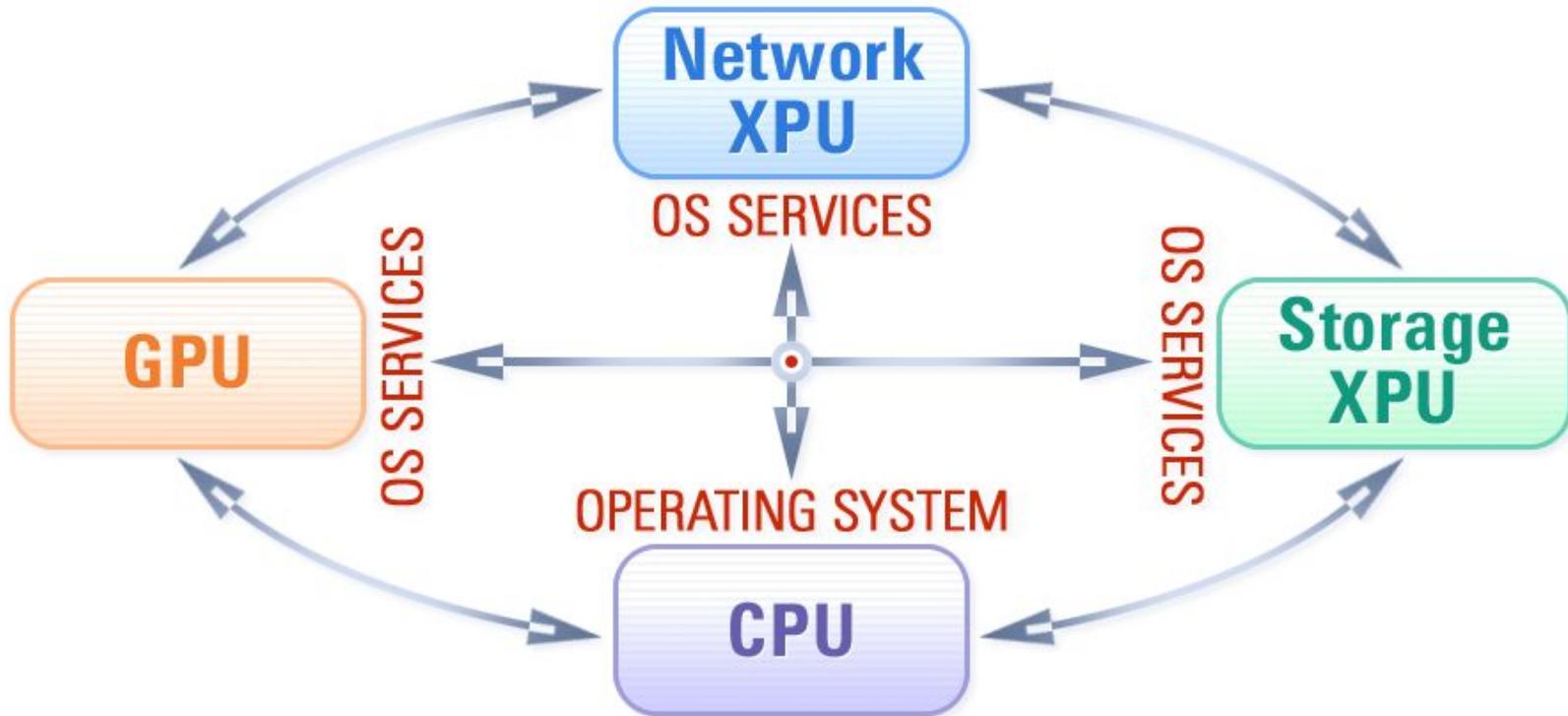
How do we program for performance?

How do we manage program execution?

How do we compile for multiple processor architectures?

How do we build Operating Systems to manage these?

# We need a new Operating System to make it work!



# Network systems: extreme speeds

Today 200Gbps

400Gbps networks are coming soon

But CPU speeds are not improving!

Network processing must be performed in a few CPU cycles! How?

Network accelerators: SmartNICs

How can we build systems that do not rely on CPUs at all?

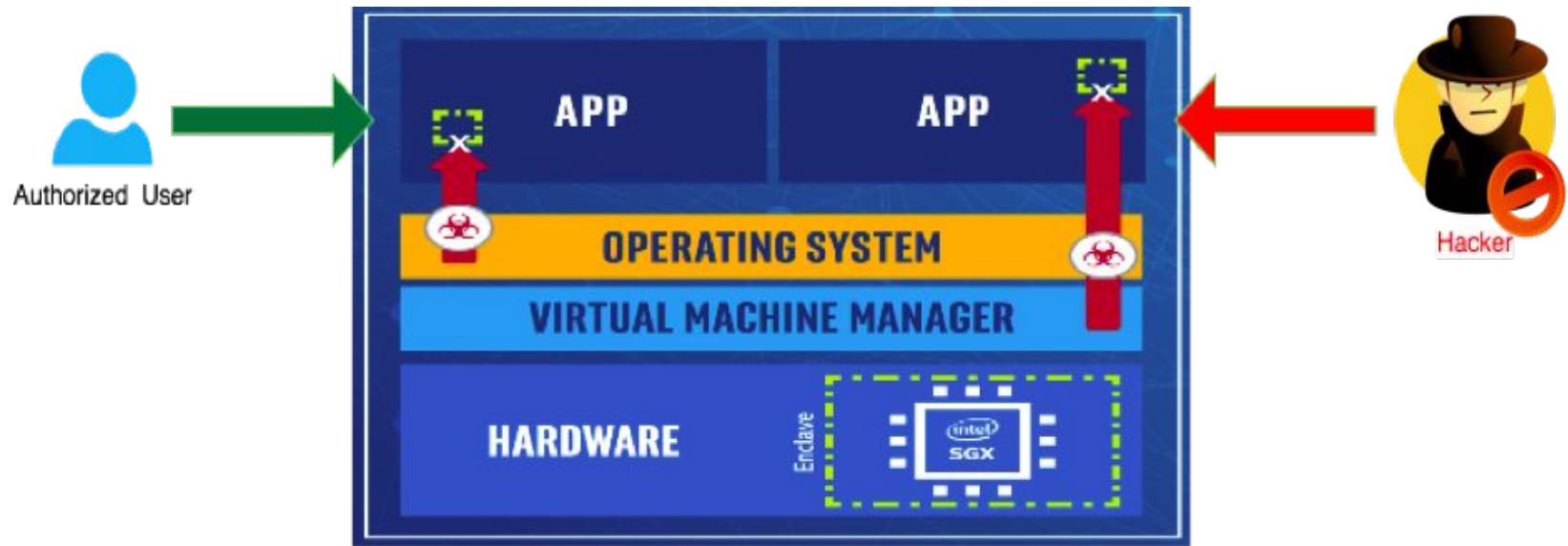
# Secure systems: how to be fast and secure?

How does NETFLIX ensure that Bob is not stealing the movie?

How do we build a system **that cannot leak** secrets?

Can we prove that a system is secure?

# Secure processor architectures



But how to achieve performance and security together?  
How “secure” is secure?

# Security is as strong as the weakest component



Breaking into trusted computing systems

# See many more at <https://acs1.group>

[Home](#)[About Us ▾](#)[Research](#)[Publications](#)[Team ▾](#)[Teaching](#)[Undergraduate Projects ▾](#)[News & Events](#)

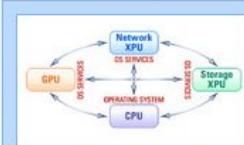
Welcome to the Accelerated Computing Systems Lab (ACSL)!

We work on a broad range of computer systems projects spanning hardware architecture, compilers, operating systems, security and privacy, high-speed networking.

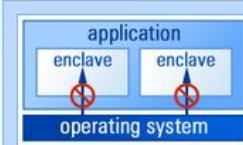
All our software is open-source and free [Image result for github image](#).

Feel passionate about building secure and fast computer systems of the future?! [Check out how to apply!](#)

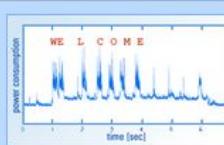
## RESEARCH

[all research areas](#)

**Accelerator-Centric Operating System**  
Accelerator-centric Operating System Architecture, OmniX,



**OS Services for Trusted Execution Environments**  
Our work facilitates the development of



**Hardware Side Channels**  
Side channels have become one of the major threats for



**GPU computing, Networking, Machine Learning, Distributed Systems**  
Playground for exploring

# But many more courses are waiting for you

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                                     |               |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Intro to C                                                                                                                                                                                                                                          |               |
| Software                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 044101                                                                                                                                                                                                                                              | Hardware (EE) |
| <ul style="list-style-type: none"><li>Intro to OS (CS/EE)</li><li>OS design and implementation (CS)</li><li>Compilers (EE)</li><li>Binary Translation (EE)</li><li>Computer networks (CS/EE)</li><li>Intro and tools in cyber security (EE)</li><li>Network security (CS)</li><li>Reverse engineering (CS)</li><li>Fundamentals of Distributed systems (EE)</li><li>Distributed and parallel programming (CS)</li><li>Object Oriented Design (CS/EE)</li><li>Functional Distributed Programming (EE)</li><li>Accelerators and accelerated systems (CS/EE)</li></ul> | <ul style="list-style-type: none"><li>Intro to computer architectures</li><li>Microprocessors</li><li>VLSI</li><li>Memristors</li><li>Ultra-fast networking</li><li>Parallel computer architecture</li><li>Advanced computer architecture</li></ul> |               |

42

Good Luck!