Midterm 2 Review

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

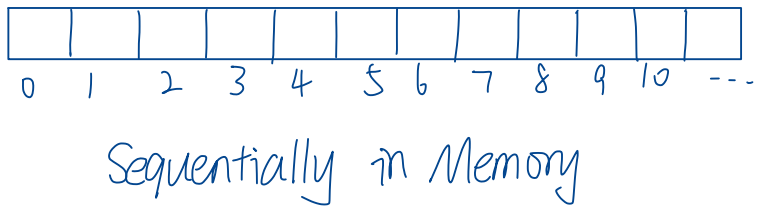
Arrays

Array: a sequence of values (collection of variables)

In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

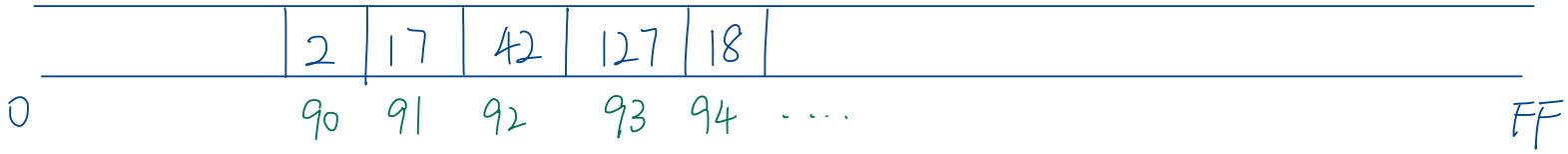
How do we store them in memory?



Arrays

arr = {2, 17, 42, 127, 18} @0x90

I assume all these are one byte so that each will fit in one of these slots.



arr[3] = 0x90 + 3 = 0x93

3 → index

0x90 → position of where our array start.

Instructions Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

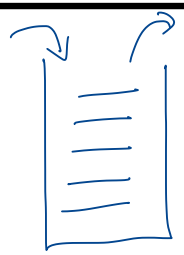
Lots of flexibility and freedom to build things that would be faster, like hyperthreading. I don't worry about on the software side.

Just make sure the code can be compiled to ISA. I can run it on hardware.

- Provides an abstraction layer between:
  - Everything computer is really doing (hardware)
  - What programmer using the computer needs to know (software)
- Hardware and Software engineers have freedom of design, if conforming to ISA
- Can change the machine without breaking any programs

The Stack

(One solution won out)



stack of plates

Stack - a last-in-first-out (LIFO) data structure

- The solution for solving this problem

rsp - Special register - the *stack* pointer

- Points to a special location in memory
- Two operations most ISAs support:
  - push - put a new value on the stack
  - pop - return the top value off the stack

have the address.
(the index in memory of a certain point)

The Stack: Push and Pop

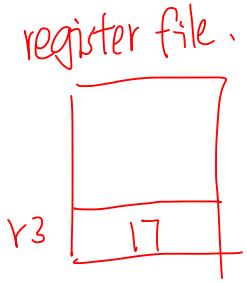
push r0

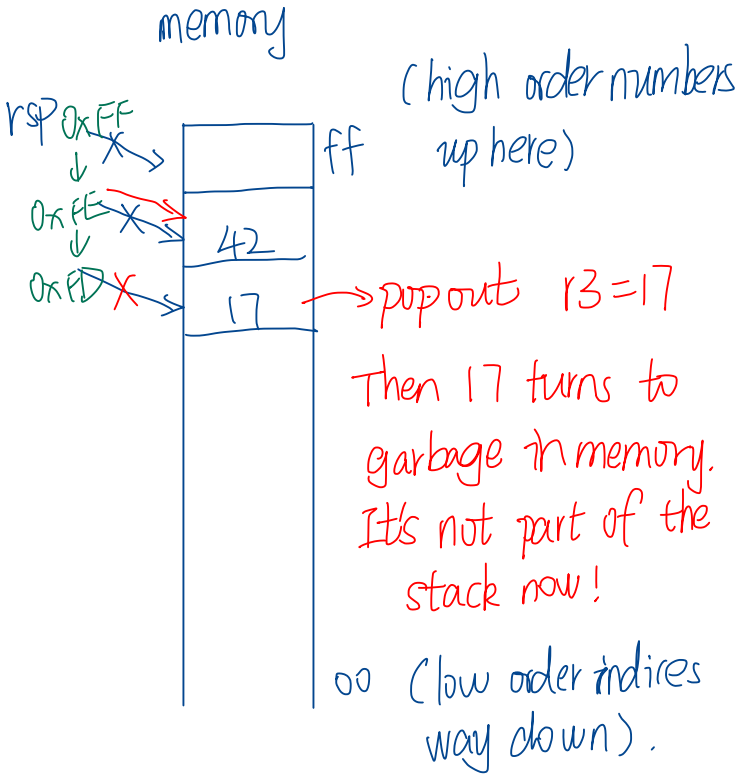
- Put a value onto the "top" of the stack
  - rsp -= 1
  - M[rsp] = r0

push 42
push 17
pop r3

pop r2

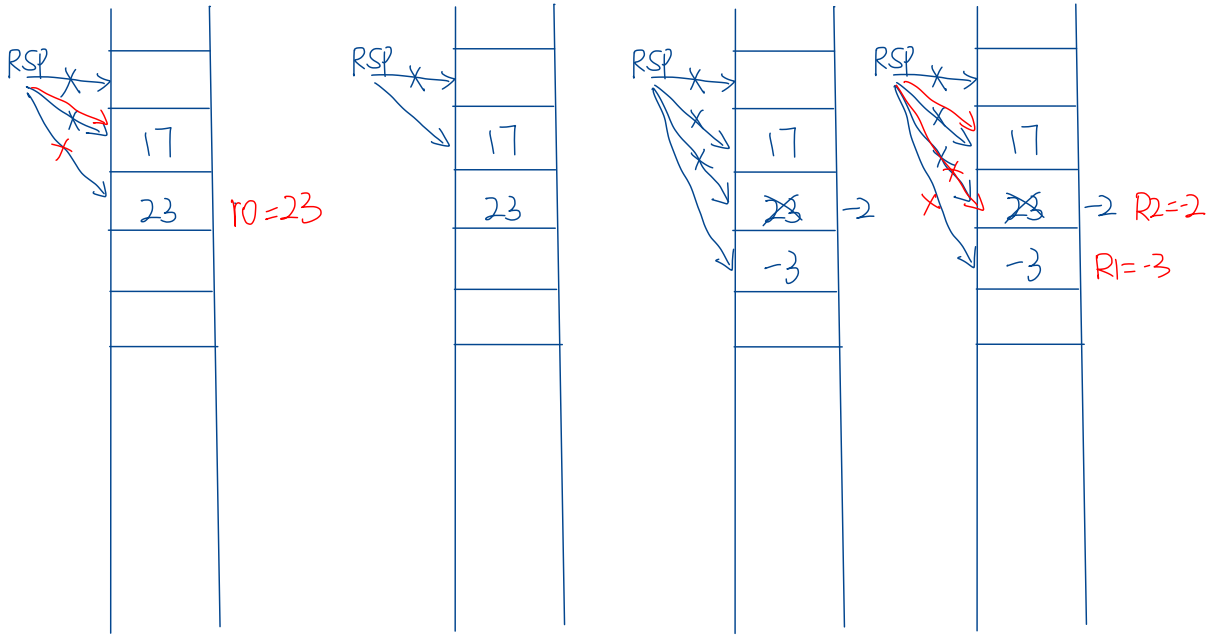
- Read value from "top", save to register
  - r2 = M[rsp]
  - rsp += 1





The Stack: Push and Pop

push (17)
push (23)
x = pop (R0)
push (-2)
push (-3)
y = pop (R1)
z = pop (R2)



Our Hardware Backdoor

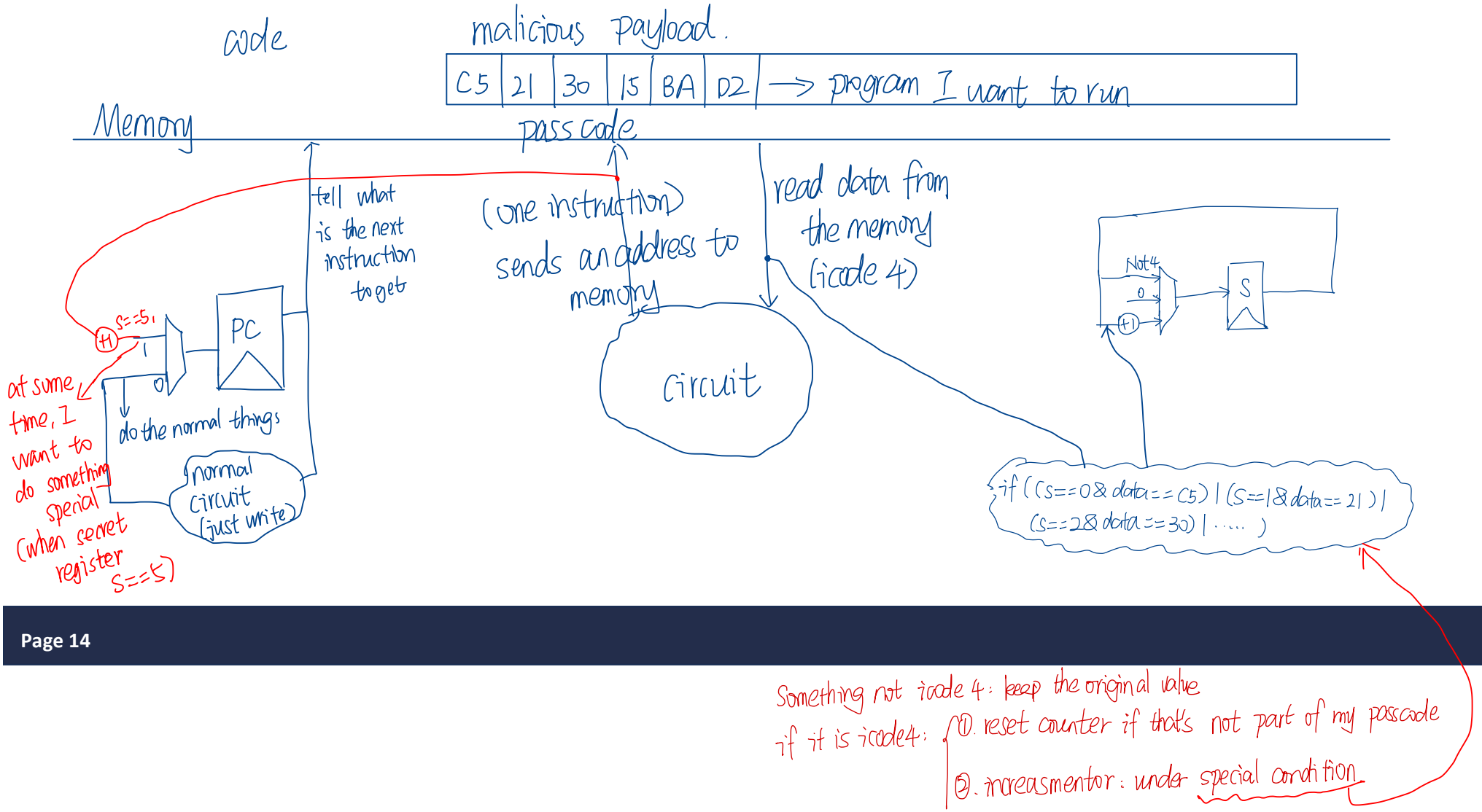
Our backdoor will have 2 components

- Passcode: need to recognize when we see the passcode
- Program: do something bad when I see the passcode

→ The hardware needs to recognize it when it appears.
(Like secret knock on a door, if you hear that pattern, you know someone special wants in.

→ Once the hardware recognizes the passcode, it executes some hidden or harmful behavior.

Our Hardware Backdoor



Our Hardware Backdoor

Will you notice this on your chip?

- Modern chips have **billions** of transistors
- We're talking adding a few hundred transistors

Backdoors

Backdoor: secret way in to do new unexpected things

- Get around the normal barriers of behavior
- Ex: a way in to allow me to take complete control of your computer

Exploit - a way to use a vulnerability or backdoor that has been created

- Our exploit today: a **malicious payload**
  - A passcode and program
  - If it ever gets in memory, run my program regardless of what you want to do

It's all bytes

Memory, Code, Data... It's all bytes!

- **Enumerate** - pick the meaning for each possible byte
- **Adjacency** - store bigger values together (sequentially)
- **Pointers** - a value treated as address of thing we are interested in

You've seen all 3 of these already.

Enumerate

Enumerate - pick the meaning for each possible byte

Assign meaning to this byte.

What is 8-bit 0x54?

Unsigned integer	eighty-four
Signed integer	positive eighty-four
Floating point w/ 4-bit exponent	twelve
ASCII	capital letter T: `T`
Bitvector sets	The set {2, 3, 5}
Our example ISA	Flip all bits of value in r1

Adjacency

Adjacency - store bigger values together (sequentially)

- An array: build bigger values out of many copies of the same type of small values
  - Store them next to each other in memory
  - Arithmetic to find any given value based on index

We know: ① The address of the first element. (addr)
② The index of that element (i)
Then the address of that element : addr + (i * size_of_each_element).

Pointers

Pointers - a value treated as address of thing we are interested in

- A value that really points to another value
- Easy to describe, hard to use properly
- We'll be talking about these a lot in this class!

Pointers

Pointers - a value treated as address of thing we are interested in

- Give us strange new powers (represent more complicated things), e.g.,
  - Variable-sized lists
  - Values that we don't know their type without looking
  - Dictionaries, maps

Those 3 things, we combine them all together. And this is kind of how we're storing the data in the memory.

Programs Use These!

How do our programs use these?

- Enumerated icodes, numbers
- Ajacently stored instructions (PC+1)
- Pointers of where to jump/goto (addresses in memory)

64-bit Machines

64-bit machine: The registers are 64-bits

- i.e., r0, but also PC

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? 256 Bytes
- Late 70s - 16 bits: 65536 Bytes
- 80s - 32 bits: ≈ 4 billion bytes
- Today's processors - 64 bits: 2^{64} addresses (2^{64} indices to memory)

A Challenge

There is a disconnect:

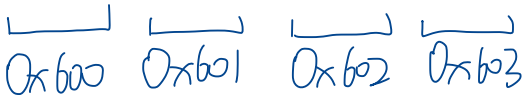
- Registers: 64-bits values
- Memory: 8-bit values (i.e., **1 byte** values)
  - Each address addresses an 8-bit value in memory
  - Each address points to a 1-byte slot in memory
- How do we store a 64-bit value in an 8-bit spot?

Rules

0x | 00 | AB | CD | EF (4 bytes).

Rules to break "big values" into bytes (memory)

1. Break it into bytes
2. Store them adjacently
3. Address of the overall value = smallest address of its bytes
4. Order the bytes
   - If parts are ordered (i.e., array), first goes in smallest address
   - Else, hardware implementation gets to pick (!!)
     – Little-endian
     – Big-endian



starting from an address and going up

Ordering Values

0x|00|AB|CD|EF

Little-endian

- Store the low order part/byte first
- Most hardware today is little-endian

EF (0x600) CD (0x601) AB (0x602) 00 (0x603)

Big-endian

- Store the high order part/byte first

00 (0x600) AB (0x601) CD (0x602) EF (0x603)

Why we want to talk about 2 ways?
Because people decided to do different things.
We write 00ABCDEF, but we calculate from F to 0,

Maybe that's the reason for processors to see EF first?

Example

array of 2 numbers, each number should use 2 bytes.

Store [0x1234, 0x5678] at address 0xF00

	address	little endian	big endian
0x1234	0xF00	34	12
	0xF01	12	34
0x5678	0xF02	78	56
	0xF03	56	78

Assembly

General principle of all **assembly languages**

- Code (text, not binary!)
- 1 line of code = 1 machine instruction
- One-to-one reversible mapping between binary and assembly
  - We do not need to remember binary encodings!
  - A program will turn text to binary for us!

- ISA is like the grammar and vocabulary of a language.
- Assembly code is a sentence written in that language.

Assembly

Features of assembly

- Automatic addresses - use **labels** to keep track of addresses
  - Assembler will remember location of labels and use where appropriate
  - Labels will not exist in machine code
- Metadata - data about data (extra information)
  - Data that helps turn assembly into code the machine can use
- As complicated as machine instructions
  - There are a lot of instructions, and it is one-to-one!

(.text .data .byte)

It's going to replace them with the actual addresses when it builds the binary that we're going to run.

Assembly

General principle of all **assembly languages**

- Code (text, not binary!)
- 1 line of code = 1 machine instruction
- One-to-one reversible mapping between binary and assembly
  - We do not need to remember binary encodings!
  - A program will turn text to binary for us!

- ISA is like the grammar and vocabulary of a language.
- Assembly code is a sentence written in that language.

Assembly

Features of assembly

- Automatic addresses - use **labels** to keep track of addresses
  - Assembler will remember location of labels and use where appropriate
  - Labels will not exist in machine code
- Metadata - data about data (extra information)
  - Data that helps turn assembly into code the machine can use
- As complicated as machine instructions
  - There are a lot of instructions, and it is one-to-one!

(.text .data .byte)

It's going to replace them with the actual addresses when it builds the binary that we're going to run.

AT&T x86-84 Assembly

`instruction source, destination`

- Instruction followed by 0 or more operands (arguments)
- 4 types of operands:
  - Number (immediate value): `$0x123`
  - Register: `%rax`
  - Address of memory: `(%rax)` or 24 or `labelname`
  - Value at an address in memory: `(%rax)` or 24 or `labelname`

(typically we will not see more than 2)

lea loading the addresses

In most of the cases, we are doing something using the value. Except for

AT&T x86-84 Assembly

`mylabelname:` end with a colon

- Label - remember the address of next thing to use later

`.something something` start with a dot

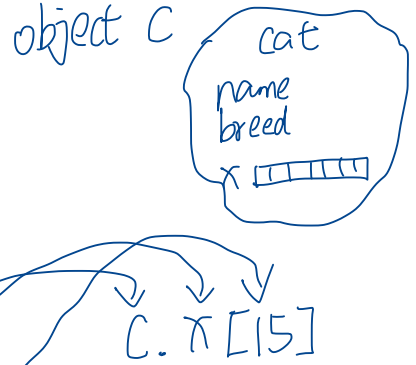
- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

`// we can have comments!`

Addressing Memory

`2130(%rax, %rsp, 8)`

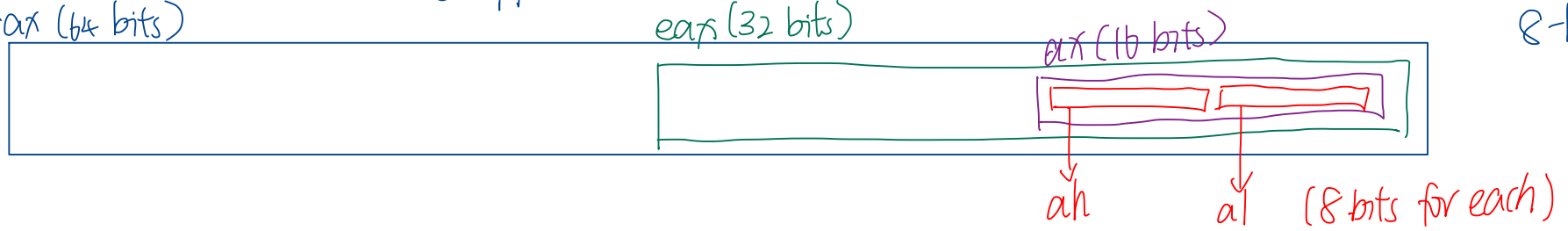
- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `2130 + %rax + (%rsp * 8)`
- Common usage from this example:
  - `rax` - address of an object in memory
  - 2130 - offset of an array into the object
  - `rsp` - index into the array
  - 8 - size of the values in the array (used to calculate the offset)
- Don't need all parts: (%rax) or 4(%rax)
- This is all one operand (one memory address)



If I don't have all the pieces, it will calculate what it can.

Registers

rax is a 64-bit register (supposed to be backwards compatible with x86 (32-bit), 16-bit, 8-bit)



If I look at 32-bit version, it will just zero out the top 32 bits.

We'll see this with all our registers, in slightly different way.

(check the reading)

Instructions (short acronyms for what we want to do, like mov, add, and, or, xor, neg)

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move (similar for addq, subq)
  - `q` = quad word
- `movl` - 32-bit move
  - `l` = long

The instruction followed by how wide of the thing we want to do.

- There are encodings for shorter things, but we will mostly see 32- and 64-bit

More powerful than our ISA

Instructions can move/operate between memory and register

- `movq %rax, %rcx` - register to register
  - Remember our icode 0
- `movq (%rax), %rcx` - memory to register
  - Remember our icode 3
- `movq %rax, (%rcx)` - register to memory
  - Remember our icode 4
- `movq $21, %rax` - Immediate to register
  - Remember our icode 6 (b=0)

Note: at most one memory address per instruction

We cannot do memory to memory calculations.

Instructions (short acronyms for what we want to do, like mov, add, and, or, xor, neg)

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move (similar for addq, subq)
  - `q` = quad word
- `movl` - 32-bit move
  - `l` = long

The instruction followed by how wide of the thing we want to do.

- There are encodings for shorter things, but we will mostly see 32- and 64-bit

Load Effective Address

I'm not going to the memory, "lea" is a special instruction that calculates the memory address and store the memory address itself in a register.

Load effective address: `leaq 4(%rcx), %rax`

- Performs memory address calculation
- Stores address, not value at the address in memory

%rax = %rcx + 4.

Jumps

`jmp foo`

- Unconditional jump to `foo`
- `foo` is a label or memory address
- Need `jmp*` to use register value (jump to a value in a register)

Conditional jumps

- `jl`, `jle`, `je`, `jne`, `jg`, `jge`, `ja`, `jb`, `js`, `jo`
  - `jl` <, `jle` <=, `je` ==, `jne` !=, `jg` >, `jge` >=, `ja` above, `jb` below
  - `ja`, `jb`: unsigned
  - `js`: If the signed bit is set.
  - `jo`: If there's a overflow

Unlike our Toy ISA, these do not compare given register to 0

Jumps

We jump based on the result of some special registers called condition codes.

Condition codes - 4 1-bit registers set by every math operation, `cmp`, and `test`

- Result for the operation compared to 0 (if no overflow)
- Example:

```
addq $-5, %rax
// ...code that doesn't set condition codes...
je foo
```

They don't have to be back to back.

→ You can do something like move things around.

jump will be based on the most recent thing that set the condition code.

  - Sets condition codes from doing math (subtract 5 from rax)
  - Tells whether result was positive, negative, 0, if there was overflow, …
  - Then jump if the result of operation should have been = 0

Jumps: compare…

`cmpq %rax, %rdx`

- Compare checks result of -= and sets condition codes
- How `rdx - rax` compares with 0
- Be aware of ordering!
  - if `rax` is bigger, sets < flag
  - if `rdx` is bigger, sets > flag

Jumps: … and test

`testq %rax, %rdx`

- Sets the condition codes based on `rdx & rax`
- Less common

Neither save their result, just set condition codes!

test could be used to check if a register has 0 in it.

```
testq %rax, %rax
je zero_case    // if rax==0
jne nonzero_case // if rax!=0
```

Example: Loops

```
while (i < 10)
    i += 1
```

top: ← lable
// check !condition, jump out
if(i>=10) goto end
    i+=1;
// jump back to condition
go to top;
end: ← lable

```
main:                          → lable
    movq $0, %rax              // we set rax=0 for int i=0;
loop:
    cmp $10, %rax              // rax-10 = ?  { if rax<10, we got negtive, then do loop body.
                                             { if rax >=10, we got positive or 0, then jump out the loop.
    jge after
    addq $1, %rax
    jmp loop
after:
    retq                       // return with a "q" because we're working with a 64 bit thing. (pop a 8-byte address and jump back to caller)
```

Function Calls: Calling Conventions

`callq myfun`

- Push return address, then jump to myfun
- Convention: Store arguments in registers and stack before call
  - First 6 arguments (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - If more arguments, pushed onto stack (last to first)

`retq`

- Pop return address from stack and jump back
- Convention: store return value in `rax` before calling `retq`

This is similar to our Toy ISA's function calls in homework 4

More conventions, check readings.

Calling Conventions: Registers

The function I'm running currently and the function that I call are both sharing the same registers.

Calling conventions - recommendations for making function calls

Why? Caller and callee share the same registers.

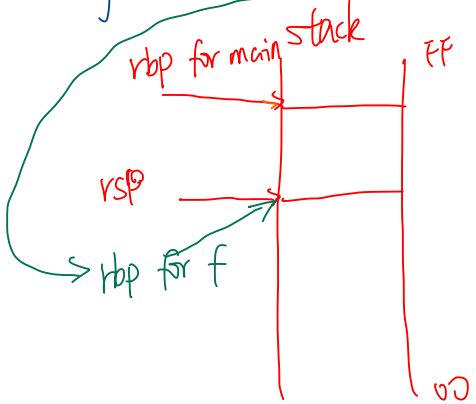
- Where to put arguments/parameters for the function call?
- Where to put return value? in `rax` before calling `retq`
- What happens to values in the registers?
  - **Callee-save** - The function should ensure the values in these registers are unchanged when the function returns
    - `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`
  - **Caller-save** - Before making a function call, save the value, since the function may change it

① push the old values before calling ② pop the values before returning.

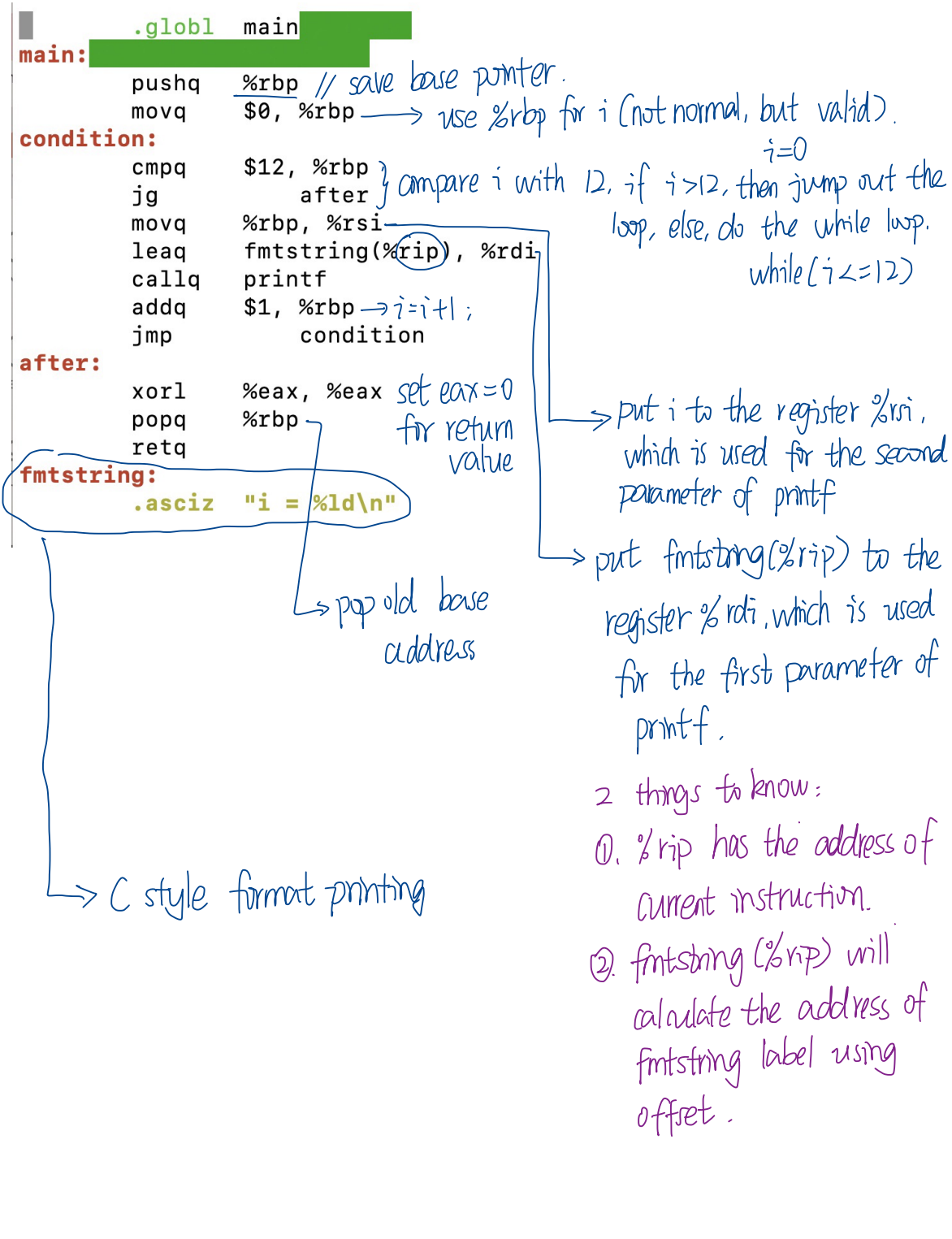
Example: Functions

```
f(x,y):
    ...
    ...
    return 4

...
z = f(2,5)
```

```
int f(int x, int y) {
    return 4;
}

int main() {
    int z = f(2,5);
}
```




```
.global f
f:
    pushq %rbp          // store old base pointer
    movq %rsp, %rbp     // create a new stack for f
    movl $4, %eax       // put return value 4 to %eax
    popq %rbp           // pop old base pointer (for main)
    retq                // return.
.global main
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $2, %edi       // put parameter 2 to %edi
    movl $5, %esi       // put 5 to %esi
    callq f
    movl %eax, -4(%rbp) // store return value to local variable z.
```


.globl main
main:
pushq %rbp // save base pointer.
movq $0, %rbp → use %rbp for i (not normal, but valid).
i=0
condition:
cmpq $12, %rbp } compare i with 12, if i>12, then jump out the
jg after loop, else, do the while loop.
while(i<=12)
movq %rbp, %rsi
leaq fmtstring(%rip), %rdi
callq printf
addq $1, %rbp → i=i+1;
jmp condition
after:
xorl %eax, %eax set eax=0
popq %rbp for return
retq value
fmtstring:
.asciz "i = %ld\n"
put i to the register %rsi, which is used for the second parameter of printf
put fmtstring(%rip) to the register %rdi, which is used for the first parameter of printf.
pop old base address
C style format printing
2 things to know:
① %rip has the address of current instruction.
② fmtstring(%rip) will calculate the address of fmtstring label using offset.

Most Common Instructions

- `mov` - =
- `lea` - load effective address
- `call` - push PC and jump to address
- `add` - +=
- `cmp` - set flags as if performing subtract
- `jmp` - unconditional jump
- `test` - set flags as if performing &
- `je` - jump iff flags indicate == 0
- `pop` - pop value from stack
- `push` - push value onto stack
- `ret` - pop PC from the stack

Debugger

objdump -D substract | less → show by pages.
→ tool name
→ disassemble all sections.

// disassemble all sections from binary to assembly

Debugger - step through code!

- Helpful for Homework 5, 6, and when we get to C
- Experience seeing results of these instructions step-by-step
- **Please read the x86-64 summary reading!**

lldb substrat // run the executable file "substract" in debugging mode.

b main // add a breakpoint at the beginning of main function

r / run // run the code

n // run the next instruction

disassemble // disassemble the binary code to assembly code (work for the current frame-main).

re read // read the registers.

Patents and Copyright

Patents and Copyright

Remember our Toy ISA. Can we patent our ISA? Should we?

icode	b	meaning
0		`rA = rB`
1		`rA &= rB`
2		`rA += rB`
3	0	`rA = ~rA`
	1	`rA = !rA`
	2	`rA = -rA`
	3	`rA = pc`
4		`rA` = read from memory at address `rB`
5		write `rA` to memory at address `rB`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` &= read from memory at `pc + 1`
	2	`rA` += read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to 0
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

Patents and Copyright

Copyright

- "Everyone is a copyright owner. Once you create an original work and fix it, like taking a photograph, writing a poem or blog, or recording a new song, you are the author and the owner."
- from https://www.copyright.gov/what-is-copyright/

Patents and Copyright

Patent

- "Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title."
- from 35 U.S.C. 101

Common Approaches to Software

How can we get value from what we create?

- Copyright - distribute closed source software
- License Agreements (in contract law)
- Always innovate

C

C is a thin wrapper around assembly

- This is by design!
- Invented to write an operating system
  - Can write inline assembly in C
- Many other languages decided to look like C

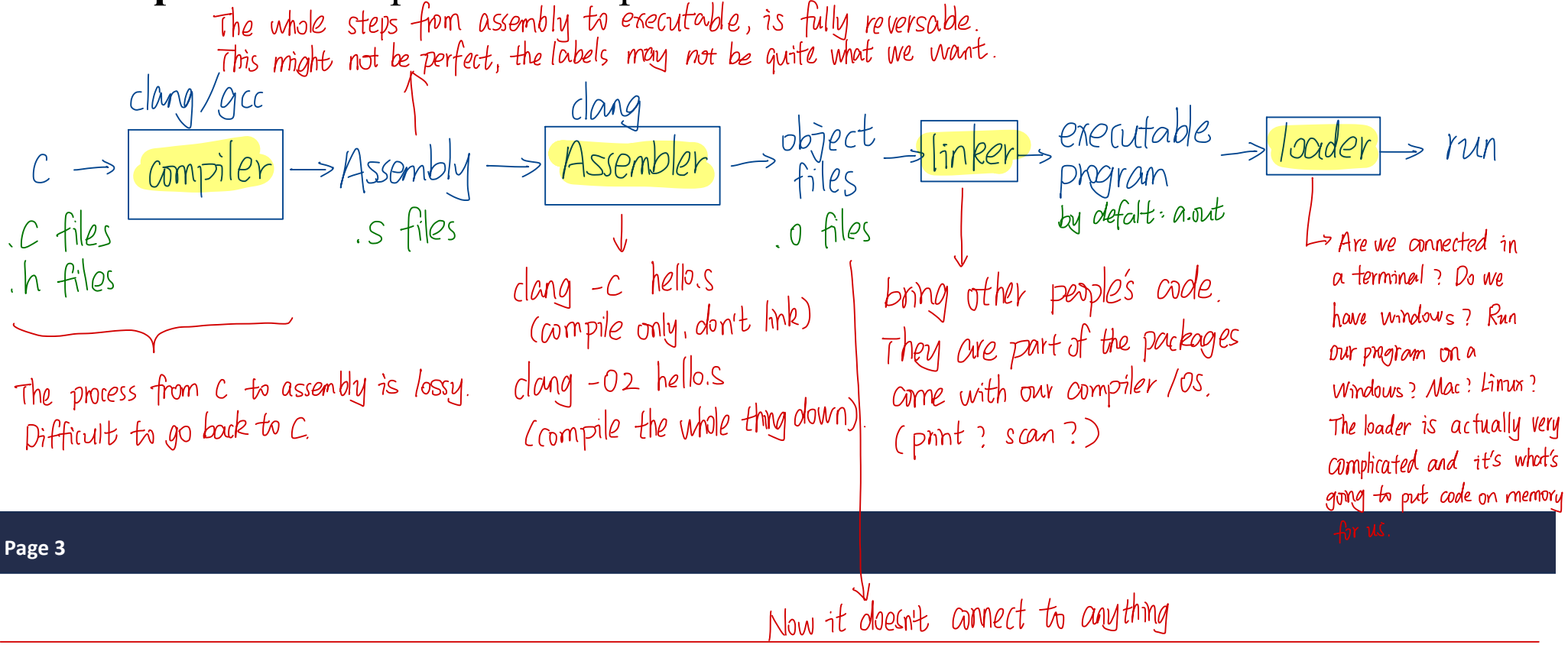
No classes / objects in C.

Compilation Pipeline

We want to bridge the gap between the assembly and C.

Turning our code into something that runs

- **Pipeline** - a sequence of steps in which each builds off the last



ls /usr/lib : lots of compiled things ready to be used.

ls /usr/lib64 | wc -l : show how many files.

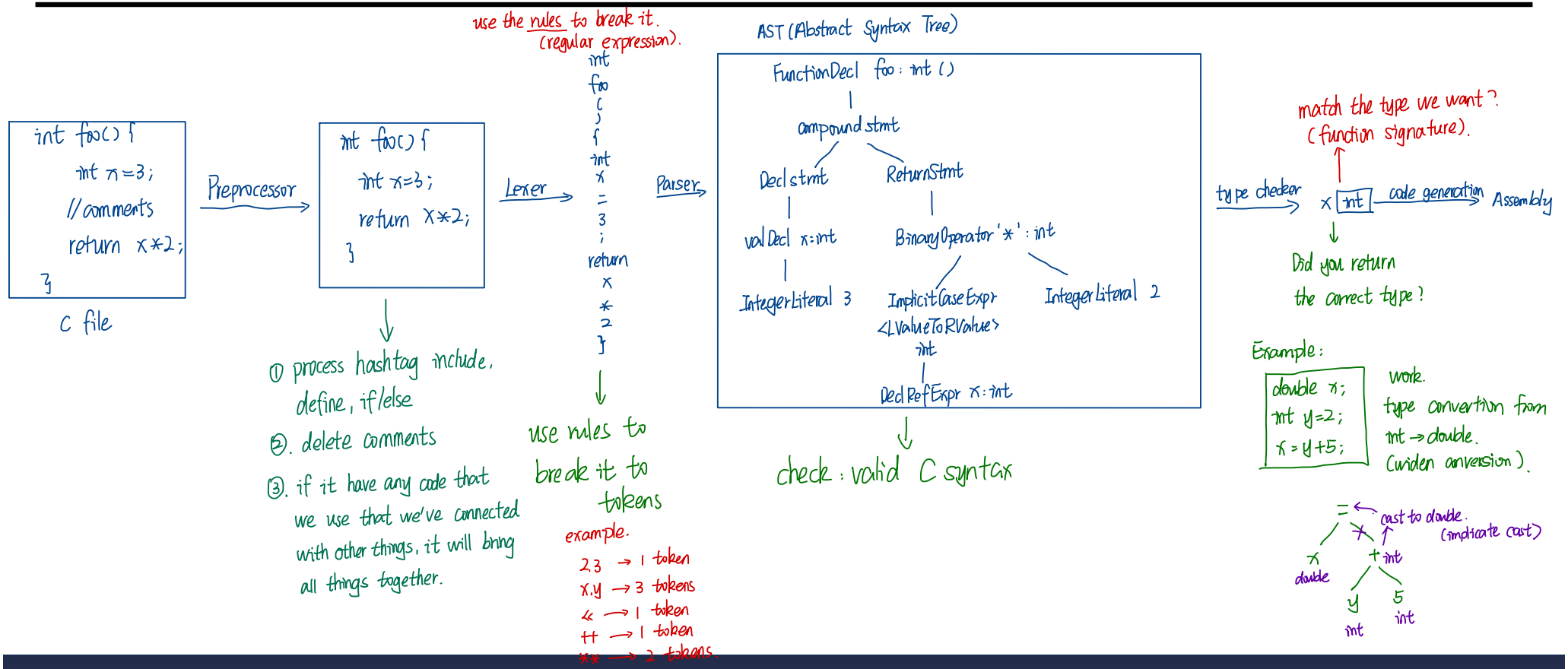
Compiling C to Assembly

Multiple stages to compile C to assembly

- Preprocess - produces C
  - C is actually implemented as 2 languages: C preprocessor language, C language
  - Removes comments, handles preprocessor directives (#)
  - `#include`, `#define`, `#if`, `#else`, …
- Lex - breaks input into individual tokens
- Parse - assembles tokens into intended meaning (parse tree)
- Type check - ensures types match, adds casting as needed
- Code generation - creates assembly from parse tree

→ has a lot of things like hashtags

Compiling C to Assembly



```c
int main() {
    return 0;
}
```

clang -O0 -c hello.c -Wno-implicit-function-declaration -o hello.o

-c → compile only, no link.

-O0 → No optimization

-Wno-implicit-function-declaration → I don't want the warning. (for example, I call a function but didn't declare it.)

llvm-objdump -d hello.o

-d → disassemble.

```
hello.o:        file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       b: 31 c0                         xorl    %eax, %eax
       d: 5d                            popq    %rbp
       e: c3                            retq
```

This is the disassemble code for main. start from address 0.

This is an unnecessary local variable initialization (a dead store), artificially introduced by compiler under -O0 for simplicity and debuggability.

1. why xorl %eax, %eax ?

① Can we use movl $0, %eax ? Yes, but xor is shorter and faster.

② Why not xorq %rax, %rax ?

The compiler is choosing to use this as an easy way — a short way — to zero out the return register. It's only two bytes to clear that register with zero. Performing a 32-bit xor on eax automatically clears the upper 32 bits to zero.

2. For different return types:

Regardless of whether the return type is int, short, long, long long. according to the x86-64 calling convention. return values are passed via the rax register. For floating-point types, such as "float" and "double", use xmm0. (Completely different hardware).

```c
int main() {
    return 0;
}

long foo(){
    return 0;
}
```

```
0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       b: 31 c0                         xorl    %eax, %eax
       d: 5d                            popq    %rbp
       e: c3                            retq
       f: 90                            nop

0000000000000010 <foo>:
      10: 55                            pushq   %rbp
      11: 48 89 e5                      movq    %rsp, %rbp
      14: 31 c0                         xorl    %eax, %eax
      16: 5d                            popq    %rbp
      17: c3                            retq
```

```c
int main() {
    puts("It's Friday!");
    return 0;
}
```

```
hello.o:        file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 48 83 ec 10                   subq    $0x10, %rsp
       8: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       f: 48 8d 3d 00 00 00 00          leaq    (%rip), %rdi
      # 0x16 <main+0x16>
      16: b0 00                         movb    $0x0, %al
      18: e8 00 00 00 00                callq   0x1d <main+0x1d>
      1d: 31 c0                         xorl    %eax, %eax
      1f: 48 83 c4 10                   addq    $0x10, %rsp
      23: 5d                            popq    %rbp
      24: c3                            retq
```

→ It calls something it doesn't know about. Because I haven't actually told it.

The linker will eventually overwrite and tell it where to actually call "puts".

```
mrq9gz@portal03:~/examples1$ clang hello.c -o hello
hello.c:2:5: error: call to undeclared function 'puts'; ISO C99 and later do not support implicit function declarations [-Wimplicit-function-declaration]
    2 |     puts("It's Friday!");
      |     ^
1 error generated.
```

```c
int puts(const char * string);

int main() {
    puts("It's Friday!");
    return 0;
}
```

manually declare puts.

Declaration → let compiler know the signature of the function.

```c
#include <stdio.h>

int main() {
    puts("It's Friday!");
    return 0;
}
```

Also works, and better

if I use "cpp hello.c | less", I can see all things in stdio.h is actually copied to the hello.c file.

cpp: c preprocessor

Data Types in C

Integer data types

Data type	Size (bits)	Size (bytes)
char	8	1
short	16	2
int	32	4
long	64	8
long long	64	8

Each has 2 versions: *signed* and *unsigned*

by default, short, int, long, long long are signed.

char is implementation-defined. { char - depends on compiler/platform
signed char
unsigned char

Example of creating a variable: unsigned long long x = 5;

Data Types in C (check "Readings → C Reference" for exponent bits and fraction bits)

Floating point

- float
- double

Data Types in C

Pointers - how C uses addresses!

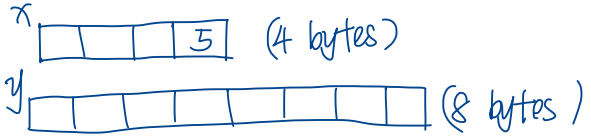
- Hold the address of a position in memory
- Need to know the kind of information stored at that location

The size of a pointer? How many bytes? — 64 bits (8 bytes) (Registers are 64 bits because we had 64-bit memory addresses)

If I want to have a pointer to an int: `int *y;`

→ the star references the fact that this is not an integer, it's a pointer.


```
int x=5;
int *y;
y=x; → doesn't work
```



`y=&x;` It works!

`*y=25;` will change x to 25.

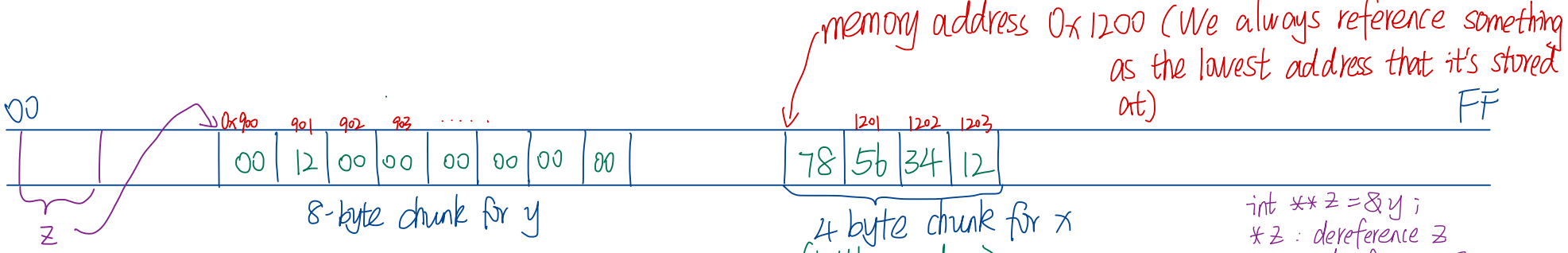
int ***z = &y; z is a pointer to a pointer to an int.
z has the address of y.

Data Types in C

later, if I say "*z", that means y.
"**z": the value of x.

Pointers - how C uses addresses!

- Hold the address of a position in memory
- Need to know the kind of information stored at that location



int x = 0x12345678

int *y;

y = &x, (I want y point to x, which means store the address of x in y).

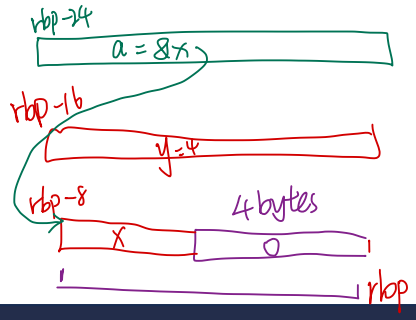
int **z = &y;
*z : dereference z
**z: dereference *z
**z = 0x00000000;
(set x to 0x00000000)
*z = 25; (Warning: you're trying to assign an integer to a pointer)
(But later, when you use it as an address, may seg Fault).

y = 0xABCDEF01; (I'm changing the pointer) → I may don't want to do this. Sometimes seg fault?

*y = 25; (When I use an asterisk, it will say is follow the pointer on y, follow the address to the place in memory that y points to and change that value)

Example

```
int main() {
    int x = 3;
    long y = 4;
    int *a = &x;
    long *b = &y;
    long z = *a;
    int w = *b;
    return 0;
}
```




```
0000000000000000 <main>:
   0:   55                          push   %rbp
   1:   48 89 e5                    mov    %rsp,%rbp
   4:   31 c0                       xor    %eax,%eax
   6:   c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
   d:   c7 45 f8 03 00 00 00        movl   $0x3,-0x8(%rbp)
  14:   48 c7 45 f0 04 00 00        movq   $0x4,-0x10(%rbp)
  1b:   00
  1c:   48 8d 4d f8                 lea    -0x8(%rbp),%rcx
  20:   48 89 4d e8                 mov    %rcx,-0x18(%rbp)
  24:   48 8d 4d f0                 lea    -0x10(%rbp),%rcx
  28:   48 89 4d e0                 mov    %rcx,-0x20(%rbp)
  2c:   48 8b 4d e8                 mov    -0x18(%rbp),%rcx
  30:   48 63 09                    movslq (%rcx),%rcx
  33:   48 89 4d d8                 mov    %rcx,-0x28(%rbp)
  37:   48 8b 4d e0                 mov    -0x20(%rbp),%rcx
  3b:   48 8b 09                    mov    (%rcx),%rcx
  3e:   89 4d d4                    mov    %ecx,-0x2c(%rbp)
  41:   5d                          pop    %rbp
  42:   c3                          retq
```

(compiler knows we are working on a 64-bit machine. I'll probably try to line everything to 8 bytes if I can. to make things faster for you)

loading the addr of x into rcx

y is treated as the same way.

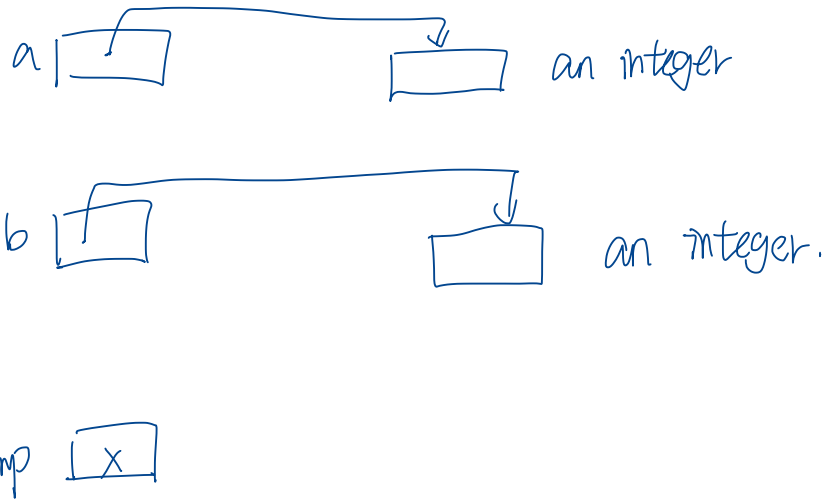
signed extend I to q, long to quad

b going from long to int. move the value of y into rcx, but I will just read out the value of ecx.

Example

Swap Example

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```



Arrays

(We pick a spot in memory, and the next so many spots are our array)

Array: 0 or more values of same type stored contiguously in memory

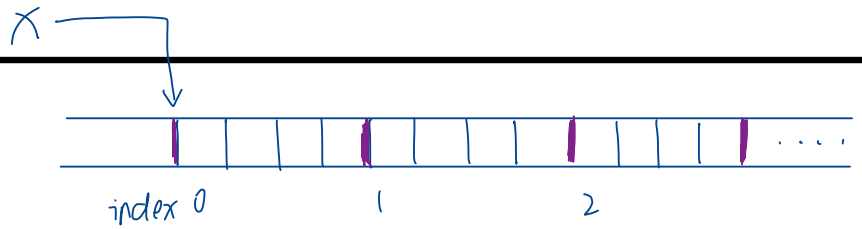
- Declare as you would use: `int myarr[100];` (100 integers in my array)
- `sizeof(myarr)` = 400 — 100 4-byte integers (How big is my array in bytes?)
- `myarr` treated as pointer to first element
- Can declare array literals:

```
int y[5] = {1, 1, 2, 3, 5}
```

(optional)

When I create an array, it actually create a pointer to the first thing in that list.

Pointers and Arrays



(dereference x will be the first element of array x)

`*x` and `x[0]` are equivalent

- Pointer to single value and pointer to first value in array
- Treat array as pointer to the first value (lowest address)
- Indexing into array: `x[n]` and `*(x+n)` → pointer arithmetic
  - If `x` is an `int *`, then `x+1` points to **next int** in memory
  - Adding 1 to pointer adds `sizeof()` the type we're pointing to

Not skip 5 bytes in memory

I know this pointer is a pointer to an integer, so I plus 5 * sizeof(int)

Arrays

(We pick a spot in memory, and the next so many spots are our array)

Array: 0 or more values of same type stored contiguously in memory

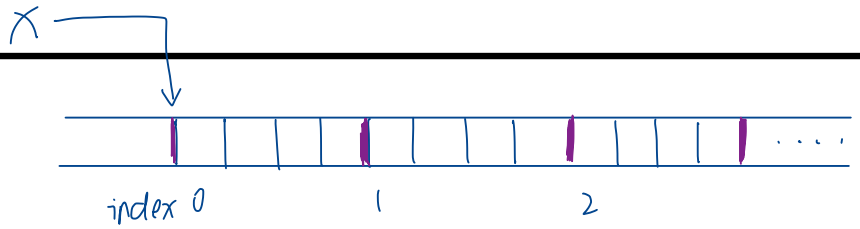
- Declare as you would use: `int myarr[100];` (100 integers in my array)
- `sizeof(myarr)` = 400 — 100 4-byte integers (How big is my array in bytes?)
- `myarr` treated as pointer to first element
- Can declare array literals:

```
int y[5] = {1, 1, 2, 3, 5}
```

(5 optional)

When I create an array, it actually create a pointer to the first thing in that list.

Pointers and Arrays



(dereference x will be the first element of array x)

`*x` and `x[0]` are equivalent

- Pointer to single value and pointer to first value in array
- Treat array as pointer to the first value (lowest address)
- Indexing into array: `x[n]` and `*(x+n)` → pointer arithmetic
  - If `x` is an `int *`, then `x+1` points to **next int** in memory
  - Adding 1 to pointer adds `sizeof()` the type we're pointing to

Not skip 5 bytes in memory

I know this pointer is a pointer to an integer, so I plus 5 * sizeof(int)

Pointers and Arrays

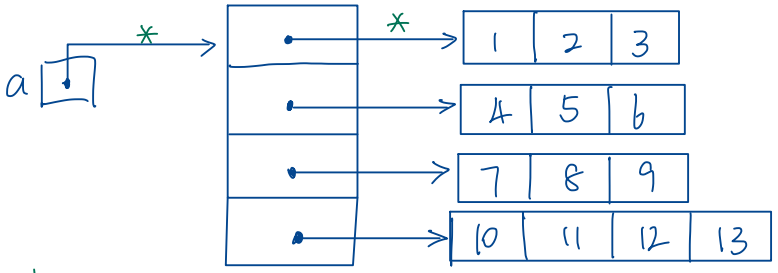
one pointer or an array of pointer?

How do I know? sizeof( )

one integer or an array of integer?

Consider: **int **a** (a pointer to a pointer to an integer)

I suppose "a" point to the first thing of an array. This array should be an array of pointers
And I suppose the pointers in this pointer array point to an array of integers.



They are just pointers to a place in memory.
It's doesn't matter how big the arrays are

```
**a ⇒ 1
a[0][0] ⇒ 1
*(a[2]) ⇒ 7
(*a)[2] ⇒ 3
a[1][3] ⇒ Something weird here. In Java, just seg fault.
a[1][0] ⇒ 4
a[0][1] ⇒ 1
*a[1] ⇒ *(a[1]) ⇒ 4
```

But in C, I can just give you the value at that place.
① Maybe seg fault if you read too far
② Maybe some number there?

} I will know something wrong when I use this value later. And if I'm writing sth, I may overwrite sth I don't know.

Pointers

sizeof(char *) == sizeof(long double *)

- All pointers are the same size: address size in underlying ISA
- Two special int types (defined using typedef)
  - `size_t` - integer the size of a pointer (unsigned)
  - `ssize_t` - integer the size of a pointer (signed)
  - With our compiler and ISA, these are both variants of `long`

sizeof() returns a value of type size_t

→ These represent integers with the same size as a pointer.

Pointers

Consider the following code:

```
int x = 10;
int *y = &x;
int *z = y + 2;
long w = ((long)z) - ((long)y);
```

We suppose x lives at address 1000

y point to x, so y == 1000

pointer arithmetic, y points to an integer, so y+2 means go forward 2 ints in memory. which is 8 bytes since y was 1000. so z will be 1000+8 = 1008

cast the address to long. So we treat them like plain integers. 1008-1000 = 8

Why is w = 8?

Other Types and Values

if I write numbers, they are implicitly cast to integers, which is 32 bits.
If I want to write something bnger, I need to add some suffixes.

- Literal values - integer literals are implicitly cast
  - `unsigned long very_big = 92233720368547758O8uL`
  - `u` for unsigned, `L` for long, `LL` for long long (capitalization doesn't matter, but you'll usually see "u" lowercase and "L" uppercase.)
- `enum` - named integer constants (in ascending order)
  - `enum { a, b, c, d=100, e };` (when I dealing with choices the user has to make, the easiest way to do is using integers, for easier reading. I give them some names)
    0 1 2 101
    I can change where I'm at using "="
    `int foo = e;`
    It's a type!
- `void` - a byte with no meaning or "nothing" → I can't do equals or any operations on it.
  - Pointers: `void *p` (I don't care what's there)
  - Return values: `void myfunction();` (I don't care what is in register rax).

```
float b = 123.4;
void *p = &b;
float c = *(float*)p;
```

- Casting - changing type, converting
  - Integer: zero- or sign-extend or truncate to space
  - Int to float: convert to nearby representable value
  - Float to int: truncate remainder (no rounding)

casting pointers does some weird things.
We'll talk about that later.

Structures

`struct` - Structures in C

- Act like Java classes, but no methods and all public fields
- Stores fields adjacently in memory (but may have padding)
- Compiler determines padding, use `sizeof()` to get size
- Name of the resulting type includes word `struct`

```
struct foo {
    long a;
    int b;
    short c;
    char d;
};
struct foo x;
x.b = 123;
x.c = 4;
```

the order of this list is important. It tells the compiler how to build this thing.

4+2+1=7

may be one byte of padding? I don't know.

↳ declare a variable x of type struct foo.

I can set directly the fields.

When you create a struct in C, the compiler decides where to place each field in memory. It tries to align each field on an address that's a multiple of it's natural size, to make the CPU access faster.

A struct in a struct?

Yes, just set a struct as one field of another struct.

Structures

`struct` - Structures in C

- Act like Java classes, but no methods and all public fields
- Stores fields adjacently in memory (but may have padding)
- Compiler determines padding, use `sizeof()` to get size
- Name of the resulting type includes word `struct`

```
struct foo {
    long a;
    int b;
    short c;
    char d;
};
struct foo x;
x.b = 123;
x.c = 4;
```

the order of this list is important. It tells the compiler how to build this thing.

4+2+1=7 may be one byte of padding? I don't know.

↳ declare a variable x of type struct foo.

I can set directly the fields.

When you create a struct in C, the compiler decides where to place each field in memory. It tries to align each field on an address that's a multiple of it's natural size, to make the CPU access faster.

A struct in a struct?

Yes, just set a struct as one field of another struct.

Structure Literals

We can use struct literals just like we saw with the array literals.

```
struct a {
    int b;
    double c;
};

/* Both of the following initialize b to 0 and c to 1.0 */
struct a x = { 0, 1.0 };
struct a y = { .b = 0, .c = 1.0 };
```

"a" is a part of the type, not a part of the name.

→ must give values to fields in order.

→ you can use equal notation to give them values in other orders.

struct a z; ⇒ If I don't give any values, just allocate the memory and whatever is in memory is whats going to be there. I don't know what they are.

typedef

`typedef` - give new names to any type!

typedef unsigned long size_t;
typedef unsigned long address_t;

- Fairly common to see several names for same data type to convey intent
- Ex: `unsigned long` may be `size_t` when used in sizes
- Examples:

```
typedef int Integer;
Integer x = 4;
typedef double ** dpp;
```

Integer becomes another name for int.

dpp ptr; (Using typedef can make complex pointer types look simpler)

- Used with *anonymous structs*:

```
typedef struct { int x; double y; } foo;
foo z = { 42, 17.4 };
```

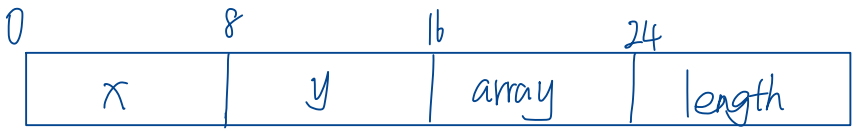
struct { ... } defines an anonymous struct (no struct name given).
The typedef immediately gives it the alias foo.
You can now create variables like foo z; instead of writing struct something z;.

Struct Example

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

remember we need this ";" at the end!

Assuming I have no padding because all of these are 64 bits



foo a;

a.array ⇒ "a+16" to get me to the array.

Struct Example

Sum2 gets a pointer, it must dereference to access fields.

arg->x is same as (*arg).x

```
long sum2(foo *arg) {
    long ans = arg->x;
    for(long i = 0; i < arg->length; i += 1)
        ans += arg->y * arg->array[i];
    return ans;
}
```

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

```
sum2:
    movq    (%rdi), %rax        ans = arg->x
    movq    24(%rdi), %r8       r8 = arg->length
    testq   %r8, %r8            performs bitwise AND.
    jle     .LBB1_3             if negative, jump to .LBB1_3
    movq    8(%rdi), %rdx       rdx = arg->y
    movq    16(%rdi), %rsi      rsi = arg->array
    xorl    %edi, %edi          i=0
.LBB1_2:
    movq    (%rsi,%rdi,8), %rcx rcx = array[i]
    imulq   %rdx, %rcx          rcx = y*array[i]
    addq    %rcx, %rax          ans += .....
    incq    %rdi                i++
    cmpq    %rdi, %r8           compare i<length
    jne     .LBB1_2
.LBB1_3:
    retq
```

integer multiply (signed)

increment

the first parameter (arg) is passed in register %rdi

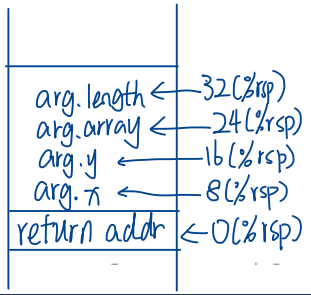
Struct Example

sum1 gets a copy of the whole struct (foo arg), it accesses fields directly on the stack.

```
long sum1(foo arg) {
    long ans = arg.x;
    for(long i = 0; i < arg.length; i += 1)
        ans += arg.y * arg.array[i];
    return ans;
}
```

Why does the compiler access the struct fields using (%rsp)+offset, instead of something like (%rdx-something)?
Modern compilers often omit it to save a register.
when you don't see %rbp, the compiler just use %rsp+offset instead.

```
sum1:
    movq    8(%rsp), %rax      ans = arg.x
    movq    32(%rsp), %r8      r8 = arg.length
    testq   %r8, %r8
    jle     LBB0_3
    movq    16(%rsp), %rdx     rdx = arg.y
    movq    24(%rsp), %rsi     rsi = arg.array
    xorl    %edi, %edi         i = 0
.LBB0_2:                       array  i  (array+i*8)
    movq    (%rsi,%rdi,8), %rcx   rcx = array[i]
    imulq   %rdx, %rcx         rcx = y*array[i]   (integer multiply (signed))
    addq    %rcx, %rax         ans += .....
    incq    %rdi               i++   (increment)
    cmpq    %rdi, %r8          compare i<length
    jne     .LBB0_2
.LBB0_3:
    retq
```




```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

Switch

```
void describe(int age) {
    switch (age) {
        case 1:
            puts("You're one.");
            break;
        case 2:
            puts("You're two.");
            break;
        case 4:
            puts("You're four.");
        case 5:
            puts("You're four or five.");
            break;
        default:
            puts("You're not 1, 2, 4 or 5!");
    }
}
```

```
    .text
    .globl  describe
describe:
    pushq   %rax
    movl    %edi, %eax
    addl    $-1, %eax
    cmpl    $4, %eax
    ja  Label5
    leaq    Text0, %rdi
    leaq    JumpTable, %rcx
    movslq  (%rcx,%rax,4), %rax
    addq    %rcx, %rax
    jmpq    *%rax
Label2:
    leaq    Text1, %rdi
    jmp Label6
Label5:
    leaq    Text4, %rdi
    jmp Label6
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
```

```
Label6:
    xorl    %eax, %eax
    callq   puts
    popq    %rax
    retq
```

```
    .section    .rodata
JumpTable:
    .long   Label6-JumpTable
    .long   Label2-JumpTable
    .long   Label5-JumpTable
    .long   Label3-JumpTable
    .long   Label4-JumpTable

Text0:
    .asciz  "You're one."
Text1:
    .asciz  "You're two."
Text2:
    .asciz  "You're four."
Text3:
    .asciz  "You're four or five."
Text4:
    .asciz  "You're not 1, 2, 4 or 5!"
```

It looks like a bunch of labels. And it's going to work exactly like we've seen labels. Where I jumped to a label and then I start executing code.

Not like a function: I go there, run things, then return.

Not like if statement: I do a bunch of conditional jumps.

One thing to note. we have to switch on an integer. That integer is going to be the index into our array of labels. (Indexing into an array of labels and picking which one we want to jump to).

```
    .globl  describe
describe:
    pushq   %rax
    movl    %edi, %eax
    addl    $-1, %eax
    cmpl    $4, %eax
    ja  Label5
    leaq    Text0, %rdi
    leaq    JumpTable, %rcx
    movslq  (%rcx,%rax,4), %rax
    addq    %rcx, %rax
    jmpq    *%rax
Label2:
    leaq    Text1, %rdi
    jmp Label6
Label5:
    leaq    Text4, %rdi
    jmp Label6
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
Label4:
    leaq    Text3, %rdi
```

first parameter (age).

The index of an array starts from 0, so we want to map 1~5 to 0~4

if index > 4, then jump to label 5

It's preparing the argument for a later call like puts (Text0).

Load the address of the jump table into %rcx.

Loads a 32-bit offset from the jump table entry indexed by %rax and sign-extends it to 64 bits. Each entry is 4 bytes long (hence the *4).

Adds the base address of the jump table to that offset.

indirect jump to the address in %rcx.

load Text1 and jump to label6.

load Text4 and jump to label6

```
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
Label4:
    leaq    Text3, %rdi
Label6:
    xorl    %eax, %eax
    callq   puts
    popq    %rax
    retq
```

empty out eax and outputs.

in assembly a long type is 32 bites.

```
    .section    .rodata
JumpTable:
    .long   Label6-JumpTable
    .long   Label2-JumpTable
    .long   Label5-JumpTable
    .long   Label3-JumpTable
    .long   Label4-JumpTable

Text0:
    .asciz  "You're one."
Text1:
    .asciz  "You're two."
Text2:
    .asciz  "You're four."
Text3:
    .asciz  "You're four or five."
Text4:
    .asciz  "You're not 1, 2, 4 or 5!"
```

jump table. Why they choose this order ?
I don't know but it doesn't matter.

read only section

Calling Functions

The C code

```
long a = f(23, "yes", 34uL);
```

follow the calling conventions if I don't know anything else about function f.

compiles to

```
movl $23, %edi
leaq label_of_yes_string, %rsi
movq $34, %rdx
callq f
# %rax is "long a" here
```

put the address of "yes" to rsi.

without respect to how `f` was defined. It is the calling convention, not the type declaration of `f`, that controls this.

Calling Functions

But, if the C code has access to the type declaration of `f`, then it might perform some implicit casting first; for example, if we declared

```
long f(double a, const char *b, double c);
```

```
long a = f(23, "yes", 34uL);
```

We need to make sure that we have the declaration of the function before we call it. So the type checker knows how the function supposed to look

then the call would be interpreted by C as having implicit casts in it:

```
long a = f((double)23, "yes", (double)34uL);
```

Calling Functions

When I call a function with floating pointer numbers, there's a whole set of registers that just yield floating point numbers.

and the arguments would be passed in floating-point registers, like so:

```
movl $23, %eax
cvtsi2sd %eax, %xmm0          # first floating-point argument
leaq label_of_yes_string, %rdi # first integer/pointer argument

movl $34, %eax
cvtsi2sd %eax, %xmm1          # second floating-point argument

callq f
# %rax is "long a" here
```

Not edi anymore. It's a general-purpose integer, not an argument.

the first floating point number register. (follow the calling conventions for floating point numbers).

cvtsi2sd convert signed integer to signed double.

For different types of the parameters, use different kinds of registers in order.

Functions Declaration

We want to know what f is so that we can call it appropriately.

```
int f(int x);
```

→ just a function header with a semicolon, no body.

just different names

- Declaration of the function — We leave it's implementation somewhere else.
- Function header
- Function signature
- Function prototype

We want this in every file that invokes `f()`

We want to know what f looks like in every single one of our C files.