

# CSCI 2021: Introduction

Chris Kauffman

*Last Updated:  
Thu Jan 28 11:38:51 AM CST 2021*

# CSCI 2021 - Logistics

## Reading

- ▶ Bryant/O'Hallaron: Ch 1
- ▶ C references: basic syntax, types, compilation

## Goals

- ▶ Basic Model of Computation
- ▶ Begin discussion of C
- ▶ Course Mechanics

## Ongoing

Due Tue 1/26 11:59pm

- ▶ Lab01: Setup, submit to Gradescope
- ▶ HW01: Basics, online Gradescope Quiz

# “Von Kauffman” Model: CPU, Memory, Screen, Program

Most computers have 4 basic, physical components<sup>1</sup>

1. A CPU which can execute instructions
2. CPU knows WHICH instruction to execute at all times
3. MEMORY where data is stored and can change
4. Some sort of Input/Output device like a SCREEN

The CPU is executes a **set of instructions**, usually called a **program**, which change MEMORY and the SCREEN

## Example of a Running Computer Program

CPU: at instruction 10:	MEMORY:	SCREEN:
> 10: set #1024 to 1801	Addr   Value	
11: set #1028 to 220	-----+-----	
12: sum #1024,#1028 into #1032	#1032   -137	
13: print #1024, "plus", #1028	#1028   12	
14: print "is", #1032	#1024   19	

---

<sup>1</sup>Of course it's a *little* more complex than this but the addage, “[All models are wrong but some are useful.](#)” applies here. This class is about asking “what is really happening?” and going deep down the resulting rabbit hole.

# Sample Run Part 1

CPU: at instruction 10:	MEMORY:	SCREEN:
> 10: set #1024 to 1801	Addr   Value	
11: set #1028 to 220	-----+-----	
12: sum #1024,#1028 into #1032	#1032   -137	
13: print #1024, "plus", #1028	#1028   12	
14: print "is", #1032	#1024   19	
CPU: at instruction 11:	MEMORY:	SCREEN:
10: set #1024 to 1801	Addr   Value	
> 11: set #1028 to 220	-----+-----	
12: sum #1024,#1028 into #1032	#1032   -137	
13: print #1024, "plus", #1028	#1028   12	
14: print "is", #1032	#1024   1801	
CPU: at instruction 12:	MEMORY:	SCREEN:
10: set #1024 to 1801	Addr   Value	
11: set #1028 to 220	-----+-----	
> 12: sum #1024,#1028 into #1032	#1032   -137	
13: print #1024, "plus", #1028	#1028   220	
14: print "is", #1032	#1024   1801	

## Sample Run Part 2

CPU: at instruction 13:

```
10: set #1024 to 1801
11: set #1028 to 220
12: sum #1024,#1028 into #1032
> 13: print #1024, "plus", #1028
14: print "is", #1032
```

MEMORY:

Addr	Value
-----+-----	
#1032	2021
#1028	220
#1024	1801

SCREEN:

CPU: at instruction 14:

```
10: set #1024 to 1801
11: set #1028 to 220
12: sum #1024,#1028 into #1032
13: print #1024, "plus", #1028
> 14: print "is", #1032
```

MEMORY:

Addr	Value
-----+-----	
#1032	2021
#1028	220
#1024	1801

SCREEN:

1801 plus 220

CPU: at instruction 15:

```
10: set #1024 to 1801
11: set #1028 to 220
12: sum #1024,#1028 into #1032
13: print #1024, "plus", #1028
14: print "is", #1032
> 15: ....
```

MEMORY:

Addr	Value
-----+-----	
#1032	2021
#1028	220
#1024	1801

SCREEN:

1801 plus 220  
is 2021

# Observations: CPU and Program Instructions

- ▶ Program instructions are usually small, simple operations:
  - ▶ Put something in a specific memory cell using its **address**
  - ▶ Copy the contents of one cell to another
  - ▶ Do arithmetic (add, subtract, multiply, divide) with numbers in cells and specified constants like 5
  - ▶ Print stuff to the screen
- ▶ The CPU keeps track of which instruction to execute next
- ▶ In many cases after executing it moves ahead by one instruction but you all know **jumping** around is also possible
- ▶ This program is in **pseudocode**: not C or Java or Assembly...
- ▶ Pseudocode can have almost anything in it so long as a human reader understands the meaning
- ▶ Real machines require more precise languages to execute as they are (still) much dumber than humans

# Observations: Screen and Memory

## Screen versus Memory

- ▶ Nothing is on the screen until it is explicitly print-ed by the program
- ▶ Normally you don't get to see memory while the program runs
- ▶ **Good programmers** can quickly form a mental picture of **what memory looks like** and draw it when needed
- ▶ You will draw memory diagrams in this class to develop such mental models

## Memory Cells

- ▶ Memory cells have Fixed **ADDRESS**  
Changeable **CONTENTS**
- ▶ Random Access Memory (RAM): the value in any memory cell can be retrieved FAST using its address
- ▶ My laptop has 16GB of memory = 4,294,967,296 (4 billion) integer boxes (!)
- ▶ Cell Address #’s never change: always cell #1024
- ▶ Cell Contents frequently change: set #1024 to 42

# Variables: Named Memory Cells

- ▶ Dealing with raw memory addresses is tedious
- ▶ Any programming language worth its salt will have **variables**: symbolic names associated with memory cells
- ▶ **You pick variable names**; compiler/interpreter automatically translates to memory cell/address

## PROGRAM ADDRESSES ONLY

CPU: at instruction 50:

```
> 50: copy #1024 to #1032
    51: copy #1028 to #1024
    52: copy #1032 to #1028
    53: print "first",#1024
    54: print "second",#1028
```

MEMORY:

Addr	Value
#1032	?
#1028	31
#1024	42

## PROGRAM WITH NAMED CELLS

CPU: at instruction 51:

```
> 50: copy x to temp
    51: copy y to x
    52: copy temp to y
    53: print "first",x
    54: print "second",y
```

MEMORY:

Addr	Name	Value
#1032	temp	?
#1028	y	31
#1024	x	42



# Correspondence of C Programs to Memory

- ▶ C programs require memory cell names to be declared with the **type of data** they will hold (*a novel idea when C was invented*).
- ▶ The equal sign (=) means  
“store the result on the right in the cell named on the left”
- ▶ Creating a cell and giving it a value can be combined

```
int x;           // need a cell named x, holds an integer
x = 42;          // put 42 in cell x
int y = 31;      // need a cell named y and put 31 in it
int tmp = x + y; // cell named tmp, fill with sum of x and y
```

## Other Rules

- ▶ C/Java compilers read whole programs to figure out how many memory cells are needed based on declarations like `int a;` and `int c=20;`
- ▶ Lines that only declare a variable do nothing except indicate a cell is needed to the compiler
- ▶ In C, uninitialized variables may have arbitrary crud in them making them dangerous to use: *we'll find out why in this course*

## Exercise: First C Snippet

- ▶ Lines starting with `//` are comments, ignored
- ▶ `printf("%d %d\n",x,y)` shows variable values on the screen

CPU: at line 50	MEMORY:	SCREEN:
> 50: int x;	Addr   Name   Value	
51: x = 42;	-----+-----+-----	
52: int y = 31;	#1032   y   ?	
53: // swap x and y (?)	#1028   x   ?	
54: x = y;	#1024	
55: y = x;		
56: printf("%d %d\n",x,y);		

With your Breakout Room colleagues:

1. Designate a **writer** who will share screen / write down answers
2. Show what memory / screen look like after running the program
3. **Correct** the program if needed: make swapping work

I will chat with one group in the main room later about their answers which will earn them an **Engagement Point**.

## Answer: First C Snippet

CPU: at line 54	MEMORY:	SCREEN:
50: int x;	Addr   Name   Value	
51: x = 42;	-----+-----+-----	
52: int y = 31;	#1032   y   31	
53: // swap x and y (?)	#1028   x   42	
> 54: x = y;	#1024	
55: y = x;		
56: printf("%d %d\n",x,y);		

CPU: at line 55	MEMORY:	SCREEN:
50: int x;	Addr   Name   Value	
51: x = 42;	-----+-----+-----	
52: int y = 31;	#1032   y   31	
53: // swap x and y (?)	#1028   x   31	
54: x = y;	#1024	
> 55: y = x;		
56: printf("%d %d\n",x,y);		

CPU: at line 57	MEMORY:	SCREEN:
50: int x;	Addr   Name   Value	31 31
51: x = 42;	-----+-----+-----	
52: int y = 31;	#1032   x   31	
53: // swap x and y (?)	#1028   y   31	
54: x = y;	#1024	
55: y = x;		
56: printf("%d %d\n",x,y);		
> 57: ...		

Clearly **incorrect**: how does one swap values properly? (fix swap\_bad.c)

## First Full C Program: swap\_main.c

```
1  /* First C program which only has a main(). Demonstrates proper
2     swapping of two int variables declared in main() using a third
3     temporary variable. Uses printf() to print results to the screen
4     (standard out). Compile run with:
5
6     > gcc swap_main.c
7     > ./a.out
8  */
9
10 #include <stdio.h>                // headers declare existence of functions
11                                // printf in this case
12 int main(int argc, char *argv[]){ // ENTRY POINT: always start in main()
13     int x;                      // declare a variable to hold an integer
14     x = 42;                      // set its value to 42
15     int y = 31;                  // declare and set a variable
16     int tmp = x;                 // declare and set to same value as x
17     x = y;                       // put y's value in x's cell
18     y = tmp;                     // put tmp's value in y's cell
19     printf("%d %d\n",x,y);       // print the values of x and y
20     return 0;                   // return from main(): 0 indicates success
21 }
```

- ▶ Swaps variables using tmp space (**exotic alternatives exist**)
- ▶ Executables always have a main() function: starting point
- ▶ Note inclusion of **stdio.h header** to declare printf() exists, allusions to C's (limited and clunky) library system

## Exercise: Functions in C, swap\_func.c

```
1 // C program which attempts to swap using a function.
2 //
3 // > gcc swap_func.c
4 // > ./a.out
5
6 #include <stdio.h>                // declare existence printf()
7 void swap(int a, int b);          // function exists, defined below main
8
9 int main(int argc, char *argv[]){ // ENTRY POINT: start executing in main()
10     int x = 42;
11     int y = 31;
12     swap(x, y);                   // invoke function to swap x/y (?)
13     printf("%d %d\n",x,y);        // print the values of x and y
14     return 0;
15 }
16
17 // Function to swap (?) contents of two memory cells
18 void swap(int a, int b){          // arguments to swap
19     int tmp = a;                  // use a temporary to save a
20     a = b;                        // a <- b
21     b = tmp;                      // b <- tmp=a
22     return;
23 }
```

Does swap() “work”? Discuss in Breakout Room and write down justification of why/why not

## Answers: The Function Call Stack and swap()

9: int main(...){	STACK: Caller main(), prior to swap()
10: int x = 42;	FRAME   ADDR   SYM   VALUE
11: int y = 31;	-----+-----+-----+-----
+<-12: swap(x, y);	main()   #2048   x   42   stack frame
13: printf("%d %d\n",x,y);	line:12   #2044   y   31   for main()
14: return 0;	-----+-----+-----+-----
V 15: }	
	STACK: Callee swap() takes control
18: void swap(int a, int b){	FRAME   ADDR   SYM   VALUE
+>-19: int tmp = a;	-----+-----+-----+-----
20: a = b;	main()   #2048   x   42   main() frame
21: b = tmp;	line:12   #2044   y   31   now inactive
22: return;	-----+-----+-----+-----
23: }	swap()   #2040   a   42   new frame
	line:19   #2036   b   31   for swap()
	#2032   tmp   ?   now active

- ▶ **Caller function** main() and **Callee function** swap()
- ▶ Caller **pushes** a stack frame onto the **function call stack**
- ▶ Frame has space for all Callee parameters/locals
- ▶ Caller tracks where it left off to resume later
- ▶ Caller copies values to Callee frame for parameters
- ▶ Callee begins executing at its first instruction

## Answers: Function Call Stack: Returning from swap()

```
9: int main(...){
10:   int x = 42;
11:   int y = 31;
12:   swap(x, y);
+>13:   printf("%d %d\n",x,y);
| 14:   return 0;
| 15: }
|
~ 18: void swap(int a, int b){
| 19:   int tmp = a;
| 20:   a = b;
| 21:   b = tmp;
+<22:   return;
| 23: }
```

STACK: Callee swap() returning

FRAME	ADDR	SYM	VALUE	
-----+-----+-----+-----				
main()	#2048	x	42	inactive
line:12	#2044	y	31	
-----+-----+-----+-----				
swap()	#2040	a	31	about to return
line:22	#2036	b	42	
	#2032	tmp	42	

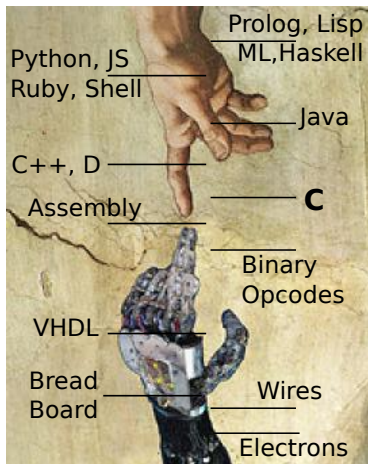
STACK: Caller main() gets control back

FRAME	ADDR	SYM	VALUE	
-----+-----+-----+-----				
main()	#2048	x	42	now
line:13	#2044	y	31	active
-----+-----+-----+-----				

- ▶ On finishing, Callee stack frame **pops** off, returns control back to Caller which resumes executing next instruction
- ▶ Callee may pass a return value to Caller but otherwise needs proper setup to alter the Caller stack frame.
- ▶ swap() does NOT swap the variables in main()

# Motivation for C

## Pure Abstraction



## Bare Metal

Source

If this were Java, Python, many others, discussion would be over:

- ▶ Provide many safety features
- ▶ Insulate programmer from hardware more

C presents most of CPU capabilities directly

- ▶ Very few safety features
- ▶ Little between programmer and hardware

*You just have to know C. Why?  
Because for all practical purposes, every computer in the world you'll ever use is a von Neumann machine, and C is a lightweight, expressive syntax for the von Neumann machine's capabilities.*  
—Steve Yegge, *Tour de Babel*



# Von Neumann Machine / Architecture

## Processing

- ▶ Wires/gates that accomplish fundamental ops
- ▶ +, -, \*, AND, OR, move, copy, shift, etc.
- ▶ Ops act on contents of memory cells to change them

## Control

- ▶ Memory address of next instruction to execute
- ▶ After executing, move ahead one unless instruction was to jump elsewhere

## Memory

- ▶ Giant array of bits/bytes so **everything** is represented as 1's and 0's, including instructions
- ▶ Memory cells accessible by address number

## Input/Output

- ▶ Allows humans to interpret what is happening
- ▶ Often special memory locations for screen and keyboard

Wait, these items seem kind of familiar...

## Exercise: C allows direct use of memory cell addresses

---

<code>&amp;x</code>	memory address of variable x
<code>int *a</code>	a stores a memory address (pointer to integer(s))
<code>*a</code>	get/set the memory pointed to by a (dereference)

---

Where are these used below?

```
1 // swap_pointer.c: swaps values using a function with pointer arguments.
2
3 #include <stdio.h>           // declare existence printf()
4 void swap_ptr(int *a, int *b); // function exists, defined below main
5
6 int main(int argc, char *argv[]){ // ENTRY POINT: start executing in main()
7     int x = 42;
8     int y = 31;
9     swap_ptr(&x, &y);          // call swap() with addresses of x/y
10    printf("%d %d\n",x,y);      // print the values of x and y
11    return 0;
12 }
13
14 // Function to swap contents of two memory cells
15 void swap_ptr(int *a, int *b){ // a/b are addresses of memory cells
16     int tmp = *a;              // go to address a, copy value int tmp
17     *a = *b;                   // copy val at addr in b to addr in a
18     *b = tmp;                  // copy temp into address in b
19     return;
20 }
```

# Swapping with Pointers/Addresses: Call Stack

```
9: int main(...){
10:   int x = 42;
11:   int y = 31;
+<12: swap_ptr(&x, &y);
| 13:   printf("%d %d\n",x,y);
| 14:   return 0;
V 15: }
```

STACK: Caller main(), prior to swap()

FRAME	ADDR	NAME	VALUE
-----+-----+-----+-----			
main()	#2048	x	42
line:12	#2044	y	31
-----+-----+-----+-----			

```
|
| 18: void swap_ptr(int *a,int *b){
+>19:   int tmp = *a;
20:   *a = *b;
21:   *b = tmp;
22:   return;
23: }
```

STACK: Callee swap() takes control

FRAME	ADDR	NAME	VALUE
-----+-----+-----+-----			
main()	#2048	x	42
line:12	#2044	y	31
-----+-----+-----+-----			
swap_ptr	#2036	a	#2048
line:19	#2028	b	#2044
	#2024	tmp	?

- ▶ Syntax &x reads “Address of cell associated with x” or just “Address of x”. Ampersand & is the address-of operator.
- ▶ Swap takes int \*a: **pointer** to integer / memory address
- ▶ Values associated with a/b are the addresses of other cells

# Swapping with Pointers/Addresses: Dereference/Use

```
9: int main(...){
10:   int x = 42;
11:   int y = 31;
12:   swap_ptr(&x, &y);
13:   printf("%d %d\n",x,y);
14:   return 0;
15: }

18: void swap_ptr(int *a,int *b){
19:   int tmp = *a; // copy val at #2048 to #2032
>20:   *a = *b;
21:   *b = tmp;
22:   return;
23: }
```

LINE 19 executed: tmp gets 42

	FRAME	ADDR	NAME	VALUE	
					<--+
	main()	#2048	x	42	<--+
	line:12	#2044	y	31	<--+
	swap_ptr	#2036	a	#2048	---+
	line:20	#2028	b	#2044	----+
		#2024	tmp	?->42	

- ▶ Syntax `*a` reads “Dereference `a` to operate on the cell pointed to by `a`” or just “Deref `a`”
- ▶ Line 19 dereferences via `*` operator:
  - ▶ Cell `#2040` (`a`) contains address `#2048`,
  - ▶ Copy contents of `#2048` (42) into `#2032` (`tmp`)

## Aside: Star/Asterisk \* has 3 uses in C

1. Multiply as in

```
w = c*d;
```

2. **Declare** a pointer as in

```
int *x; // pointer to integer(s)
```

```
int b=4;
```

```
x = &b; // point x at b
```

```
int **r; // pointer to int pointer(s)
```

3. **Dereference** a pointer as in

```
int p = *x; // x must be an int pointer
```

```
        // retrieve contents at address
```

Three different context sensitive meanings for the same symbol makes \* hard on humans to parse, a BAD move by K&R.

```
int z = *x * *y + *(p+2); // standard, 'unambiguous' C
```

The duck is ready to eat. // English is more ambiguous

# Swapping with Pointers/Addresses: Dereference/Assign

```
9: int main(...){
10:   int x = 42;
11:   int y = 31;
12:   swap_ptr(&x, &y);
13:   printf("%d %d\n",x,y);
14:   return 0;
15: }

18: void swap_ptr(int *a,int *b){
19:   int tmp = *a;
20:   *a = *b;          // copy val at #2044 (31) to #2048 (was 42)
>21:   *b = tmp;
22:   return;
23: }
```

LINE 20 executed: alters x using a

FRAME	ADDR	NAME	VALUE	
-----+-----+-----+-----				
main()	#2048	x	42->31	<--+
line:12	#2044	y	31	<- +
-----+-----+-----+-----				
swap_ptr	#2036	a	#2048	--+
line:21	#2028	b	#2044	---+
	#2024	tmp	42	

- ▶ Dereference can be used to get values at an address
- ▶ Can be used on left-hand-side of assignment to set contents at an address
- ▶ Line 20: dereference a to change contents at #2048

## Swapping with Pointers/Addresses: Deref 2

```

 9: int main(...){
10:   int x = 42;
11:   int y = 31;
12:   swap_ptr(&x, &y);
13:   printf("%d %d\n",x,y);
14:   return 0;
15: }

18: void swap_ptr(int *a,int *b){
19:   int tmp = *a;
20:   *a = *b;
21:   *b = tmp;      // copy val at #2032 (42) to #2044 (was 31)
>22:   return;
23: }
```

LINE 21 executed: alters y using b

FRAME	ADDR	NAME	VALUE	
main()	#2048	x	31	<--+
line:12	#2044	y	31->42	<- +
swap_ptr	#2036	a	#2048	---+
line:22	#2028	b	#2044	----+
	#2024	tmp	42	

- ▶ Can be used on left-hand-side of assignment to set contents at an address
- ▶ Line 21: dereference `*b = ...` to change contents at `#2044`
- ▶ Use of variable name `tmp` retrieves contents of cell associated with `tmp`

## Swapping with Pointers/Addresses: Returning

```
9: int main(...){
10:   int x = 42;
11:   int y = 31;
12:   swap_ptr(&x, &y);
+>13:   printf("%d %d\n",x,y);
| 14:   return 0;
| 15: }
|
| 18: void swap_ptr(int *a,int *b){
| 19:   int tmp = *a;
| 20:   *a = *b;
| 21:   *b = tmp;
+<22:   return;
23: }
```

LINE 22: prior to return

FRAME	ADDR	NAME	VALUE	
-----+-----+-----+-----				
main()	#2048	x	31	<--+
line:12	#2044	y	42	<- +
-----+-----+-----+-----				
swap_ptr	#2036	a	#2048	--+
line:22	#2028	b	#2044	---+
	#2024	tmp	42	

LINE 12 finished/return pops frame

FRAME	ADDR	NAME	VALUE	
-----+-----+-----+-----				
main()	#2048	x	31	
line:13	#2044	y	42	
-----+-----+-----+-----				

- ▶ swap\_ptr() finished so frame pops off
- ▶ Variables x,y in main() have changed due to use of references to them.



# Important Principle: Non-local Changes

- ▶ Pointers allow functions to change variables associated with other running functions
- ▶ Common beginner example: `scanf()` family which is used to read values from terminal or files

- ▶ Snippet from `scanf_demo.c`

```
1 int main(...){
2     int num = -1;
3     scanf("%d", &num); // addr
4     printf("%d\n", num); // val
4     return 0;
5 }
```

- ▶ See `scanf_error.c` : forgetting `&` yields great badness

`scanf()` called

FRAME	ADDR	NAME	VALUE	
main():3	#2500	num	-1	<--
scanf()	#2492	fmt	#400	
	#2484	arg1	#2500	--

`scanf()` changes contents of #2500

FRAME	ADDR	NAME	VALUE	
main():3	#2500	num	5	<--
scanf()	#2492	fmt	#400	
	#2484	arg1	#2500	--

`scanf()` returns

FRAME	ADDR	NAME	VALUE	
main():4	#2500	num	5	

# Uncle Ben Said it Best...



*All of these apply to our context..*

- ▶ **Pointers** allow any line of C programs to modify any of its data
- ▶ A BLESSING: fine control of memory → efficiency, machine's true capability
- ▶ A CURSE: opens up many **errors** not possible in langs like Java/Python which restrict use of memory

*1972 - Dennis Ritchie invents a powerful gun that shoots both forward and backward simultaneously. Not satisfied with the number of deaths and permanent maimings from that invention he invents C and Unix.*

*– A Brief, Incomplete, and Mostly Wrong History of Programming Languages*

# Beneath the C

C is a “high-level” as it abstracts away from a real machine. It must be translated to lower levels to be executed.

## Assembly Language

- ▶ Specific to each CPU architecture (Intel, etc)
- ▶ Still “human readable” but fairly directly translated to binary using Assemblers

### INTEL x86-64 ASSEMBLY

```
cmpl    $1, %ecx
jle     .END
movl    $2, %esi
movl    %ecx,%eax
cqto
idivl   %esi
cmpl    $1,%edx
jne     .EVEN
```

## Binary Opcodes

- ▶ 1's and 0's, represent the digital signal of the machine
- ▶ Codes corresponds to instructions directly understood by processor

### HEXADECIMAL/BINARY OPCODES

```
1124: 83 f9 01
1127: 7e 1e = 0111 1110 0001 1110
1129: be 02 00 00 00
112e: 89 c8
1130: 48 99
1132: f7 fe
1134: 83 fa 01
1137: 75 07
```

Looks like **fun**, right? You bet it is! Assembly coding is 1 month away...

# CSCI 2021: Course Goals

- ▶ Basic proficiency at C programming
- ▶ Knowledge of running programs in physical memory including the stack, heap, global, and text areas of memory
- ▶ Understanding of the essential elements of assembly languages
- ▶ Knowledge of the correspondence between high-level program constructs.
- ▶ Ability to use a symbolic debugger
- ▶ Basic understanding of how data is encoded in binary
- ▶ Knowledge of computer memory systems
- ▶ Basic knowledge of computer architecture