CMSC216: Introduction

Chris Kauffman

Last Updated:
Fri Jan 26 09:33:37 AM EST 2024

CMSC216 1xx-3xx: Logistics

Introductions

- Prof Kauffman: `profk@umd.edu`
- Office Hours Tue 12-1pm / Wed 1-2pm
  - Weeks 1-2: on Zoom (see Canvas link)
  - Week 3: in IRB 2226
- Slides: Linked from Canvas "Course Schedule/Materials"
- Static link: https://www.umd.edu/~profk/216/

Reading

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

Goals

- Basic Model of Computation
- Begin discussion of C

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

Most computers have 4 basic, physical components[1]

1. CPU: can execute “instructions”
2. CONTROL: CPU knows WHICH instruction to execute
3. MEMORY: data is stored and can change
4. Some sort of Input/Output device like a SCREEN (optional)

CPU understands some **set of instructions**; a sequence of instructions is a **program** that changes MEMORY and SCREEN

Example of a Running Computer Program

```
CPU: at instruction 10:                 MEMORY:               SCREEN:
> 10: set #1024 to 195                  | Addr  | Value |
  11: set #1028 to 21                   |-------+-------|
  12: sum #1024,#1028 into #1032        | #1032 |  -137 |
  13: print #1024, "plus", #1028        | #1028 |    12 |
  14: print "is", #1032                 | #1024 |    19 |
```

[1] 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 195               | Addr  | Value |
  11: set #1028 to 21                |-------+-------|
  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 195               | Addr  | Value |
> 11: set #1028 to 21                |-------+-------|
  12: sum #1024,#1028 into #1032     | #1032 |  -137 |
  13: print #1024, "plus", #1028     | #1028 |    12 |
  14: print "is", #1032              | #1024 |   195 |

CPU: at instruction 12:              MEMORY:             SCREEN:
  10: set #1024 to 195               | Addr  | Value |
  11: set #1028 to 21                |-------+-------|
> 12: sum #1024,#1028 into #1032     | #1032 |  -137 |
  13: print #1024, "plus", #1028     | #1028 |    21 |
  14: print "is", #1032              | #1024 |   195 |
```

Sample Run Part 2

```
CPU: at instruction 13:                MEMORY:              SCREEN:
  10: set #1024 to 195                 | Addr  | Value |
  11: set #1028 to 21                  |-------+-------|
  12: sum #1024,#1028 into #1032       | #1032 |   216 |
> 13: print #1024, "plus", #1028       | #1028 |    21 |
  14: print "is", #1032                | #1024 |   195 |

CPU: at instruction 14:                MEMORY:              SCREEN:
  10: set #1024 to 195                 | Addr  | Value |    195 plus  21
  11: set #1028 to 21                  |-------+-------|
  12: sum #1024,#1028 into #1032       | #1032 |   216 |
  13: print #1024, "plus", #1028       | #1028 |    21 |
> 14: print "is", #1032                | #1024 |   195 |

CPU: at instruction 15:                MEMORY:              SCREEN:
  10: set #1024 to 195                 | Addr  | Value |    195 plus  21
  11: set #1028 to 21                  |-------+-------|    is 216
  12: sum #1024,#1028 into #1032       | #1032 |   216 |
  13: print #1024, "plus", #1028       | #1028 |    21 |
  14: print "is", #1032                | #1024 |   195 |
> 15: ....
```

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 (+, -, *, /) on cells or constants
  - Print stuff to the screen
- The CPU keeps track of which instruction to execute next
- After executing an instruction, CPU advances to next instruction BUT **jumping** around to distant instructions is also possible: conditional and iterative execution
- Previous program is in **pseudocode** in which instructions can have any meaning understood by a human reader[2]
- Real machines require more precise instruction definitions as there are no smart humans to interpret them, only dumb physics to blindly execute them

[2] The pseudcode shown resembles a low-level **assembly language** rather than a high level language like C or Java

Observations: Memory Cells and the Screen

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`

Screen versus Memory

- Nothing is on the screen until it is explicitly `print`-ed by the program
- Don't get to see memory while the program runs: **print stuff while debugging programs so you can see it**
- Forming a mental model of what values are in memory and how they relate to one another is a valuable skill which we will practice, often by drawing memory explicitly

Variables are 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:           MEMORY:
> 50: copy #1024 to #1032         | Addr  | Value |
  51: copy #1028 to #1024         |-------+-------|
  52: copy #1032 to #1028         | #1032 |     ? |
  53: print "first",#1024         | #1028 |    31 |
  54: print "second",#1028        | #1024 |    42 |

PROGRAM WITH NAMED CELLS          MEMORY:
CPU: at instruction 51:           | Addr  | Name | Value |
> 50: copy x to temp              |-------+------+-------|
  51: copy y to x                 | #1032 | temp |     ? |
  52: copy temp to y              | #1028 | y    |    31 |
  53: print "first",x             | #1024 | x    |    42 |
  54: print "second",y
```

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 functions 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, not executed
- `printf("%d %d\n",x,y)` shows variable values on the screen as decimal integers

```
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 nearby colleagues:

1. Show what memory / screen look like after running the program
2. Correct the program if needed: make swapping work

I will chat with a couple folks about their answers which will earn participation credit leading to **Bonus Engagement Points**.

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_main_bad.c`)

First Full C Program: `swap_main.c`

```c
/* First C program showing a main() function. Demonstrates proper
   swapping of two int variables declared in main() using a third
   temporary variable. Uses printf() to print results to the screen
   (standard out). Compile run with:

   > gcc swap_main.c
   > ./a.out
*/

#include <stdio.h>                // headers declare existence of functions
                                  // printf in this case

int main(int argc, char *argv[]){ // ENTRY POINT: always start in main()
  int x;                          // declare a variable to hold an integer
  x = 42;                         // set its value to 42
  int y = 31;                     // declare and set a variable
  int tmp = x;                    // declare and set to same value as x
  x = y;                          // put y's value in x's cell
  y = tmp;                        // put tmp's value in y's cell
  printf("x is: %d y is: %d\n",x,y);    // print the values of x and y
  return 16;                       // return from main(): 0 indicates success
}
```

- 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`

```c
// C program which attempts to swap using a function.
//
// > gcc swap_func.c
// > ./a.out

#include <stdio.h>                     // declare existence printf()
void swap(int a, int b);               // function exists, defined below main

int main(int argc, char *argv[]){ // ENTRY POINT: start executing in main()
  int x = 42;
  int y = 31;
  swap(x, y);                          // invoke function to swap x/y (?)
  printf("%d %d\n",x,y);               // print the values of x and y
  return 0;
}

// Function to swap (?) contents of two memory cells
void swap(int a, int b){               // arguments to swap
  int tmp = a;                         // use a temporary to save a
  a = b;                               // a <- b
  b = tmp;                             // b <- tmp=a
  return;
}
```

Does `swap()` "work"? Discuss with neighbors and justify why the code works or why not

Answers: Swapping in a Function is Tricky

`swap_func.c` will not print swapped values

- If you thought the values would print swapped, you're about to learn something interesting
- If you were confident they would not print swapped but had difficulty articulating why, that's great: this class is here to give the vocab to do so
- If you knew the values wouldn't swap and also knew how to explain it well, tune in anyway as the subsequent explanation will introduce conventions used for the rest of the course

Why No Swap??

Necessitates introducing the **Function Call Stack** which is where functions store their local variables and parameters

Answers: The Function Call Stack and 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;
V  15: }
|
|  18: void swap(int a, int b){
+->19:   int tmp = a;
   20:   a = b;
   21:   b = tmp;
   22:   return;
   23: }
```

```
STACK: Caller main(), prior to swap()
| FRAME    | ADDR  | SYM | VALUE |
|----------+-------+-----+-------|
| main()   | #2048 | x   |    42 | stack frame
| line:12  | #2044 | y   |    31 | for main()
|----------+-------+-----+-------|

STACK: Callee swap() takes control
| FRAME    | ADDR  | SYM | VALUE |
|----------+-------+-----+-------|
| main()   | #2048 | x   |    42 | main() frame
| line:12  | #2044 | y   |    31 | now inactive
|----------+-------+-----+-------|
| 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 vars
- 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: }
```

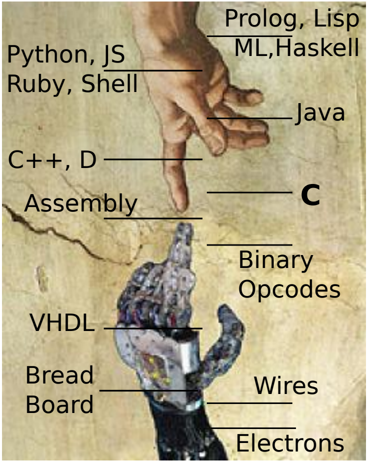
```
STACK: Callee swap() returning
| FRAME    | ADDR  | SYM | VALUE |
|----------+-------+-----+-------|
| main()   | #2048 | x   |    42 | inactive
| line:12  | #2044 | y   |    31 |
|----------+-------+-----+-------|
| swap()   | #2040 | a   |    31 | about to
| line:22  | #2036 | b   |    42 | return
|          | #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, Control returns to Caller which resumes executing next instruction
- Callee may pass a return value to Caller but otherwise does not directly affect Caller stack frame on return
- `swap()` does NOT swap the variables `x,y` in `main()`, only its own local variables `a,b`

Motivation for C

Pure Abstraction



Bare Metal

Source

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

- Provide many safety and convenience features
- Insulate programmer from hardware for ease of use

C presents many 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 (Wikip)

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

Syntax	*Meaning*
`&x`	**Address of**: memory address of variable `x`
`int *a`	**Pointer Variable**: `a` stores a memory address
`*a`	**Dereference**: get/set the value pointed to by `a`

Where/how are these used in the code below?

```c
// swap_pointer.c: swaps values using a function with pointer arguments.

#include <stdio.h>                  // declare existence printf()
void swap_ptr(int *a, int *b);      // function exists, defined below main

int main(int argc, char *argv[]){ // ENTRY POINT: start executing in main()
  int x = 42;
  int y = 31;
  swap_ptr(&x, &y);                 // call swap() with addresses of x/y
  printf("%d %d\n",x,y);            // print the values of x and y
  return 0;
}

// Function to swap contents of two memory cells
void swap_ptr(int *a, int *b){      // a/b are addresses of memory cells
  int tmp = *a;                     // go to address a, copy value int tmp
  *a = *b;                          // copy val at addr in b to addr in a
  *b = tmp;                         // copy temp into address in b
  return;
}
```

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

```
STACK: Caller main(), prior to swap()
| FRAME    | ADDR  | NAME | VALUE |
|----------+-------+------+-------|
| main()   | #2048 | x    |    42 |
| line:12  | #2044 | y    |    31 |
|----------+-------+------+-------|

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(...){                    LINE 19 executed: tmp gets 42
10:   int x = 42;                     | FRAME    | ADDR  | NAME | VALUE |
11:   int y = 31;                     |----------+-------+------+-------|
12:   swap_ptr(&x, &y);               | main()   | #2048 | x    |    42 |<-+
13:   printf("%d %d\n",x,y);          | line:12  | #2044 | y    |    31 |<-|+
14:   return 0;                       |----------+-------+------+-------|  ||
15: }                                 | swap_ptr | #2036 | a    | #2048 |--+|
                                      | line:20  | #2028 | b    | #2044 |---+
18: void swap_ptr(int *a,int *b){     |          | #2024 | tmp  | ?->42 |
19:   int tmp = *a; // copy val at #2048 to #2024
>20:  *a = *b;
21:   *b = tmp;
22:   return;
23: }
```

- Syntax `*a` reads "Dereference a to operate on the cell pointed to by a" or just "Deref a"
- Line 19 dereferences via `*` operator:
  - Cell #2036 (a) contains address #2048,
  - Copy contents of #2048 (42) into #2024 (`tmp`)

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 |
```

- Pointer Deref on Right Side **fetches** a value from a pointer location
- Pointer Deref on Left Side **stores** a value at a pointer location
- Line 20: Deref on both Left and right side of assignment
  - a and b contain pointers, not changed
  - x and y are pointed at, can change

Swapping with Pointers/Addresses: Deref 2

```
  9: int main(...){                     LINE 21 executed: alters y using b
 10:   int x = 42;                      | FRAME    | ADDR  | NAME | VALUE |
 11:   int y = 31;                      |----------+-------+------+-------|
 12:   swap_ptr(&x, &y);                | main()   | #2048 | x    |    31 |<-+
 13:   printf("%d %d\n",x,y);           | line:12  | #2044 | y    |31->42 |<-|+
 14:   return 0;                        |----------+-------+------+-------|  ||
 15: }                                  | swap_ptr| #2036 | a    | #2048 |--+|
                                        | line:22  | #2028 | b    | #2044 |---+
 18: void swap_ptr(int *a,int *b){      |          | #2024 | tmp  |    42 |
 19:   int tmp = *a;
 20:   *a = *b;
 21:   *b = tmp;     // copy val at #2024 (42) to #2044 (was 31)
>22:   return;
 23: }
```

- Line 21: dereference on left-hand side

  `*b = ...`

  stores new value at address #2044
- Use of variable **bare name** always retrieves value it that cell
  - `tmp` retrieves an `int` like 42
  - `a` retrieves a pointer like #2048

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.

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

1. Multiply numbers as in

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

2. **Declare** a pointer variable 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 variable 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
```

Some Common Examples and Errors

- Learning syntax and semantics of pointers requires some practice, get started with below examples
- Won't go through these in much detail YET but over next couple weeks will discuss at length

```c
// pointer_examples.c
// 1: proper pointer assignment
int  a1 = 11;
int *p1 = &a1;   // cool
int  b1 = 55;
p1 = &b1;        // cool

// 2: improper pointer assignment
int  a2 = 13;
int *p2 = a2;    // ERROR

// 3: proper pointer copying
int  a3 = 15;
int *p3 = &a3;
int *q3 = p3;    // cool
```

```c
// 4: proper pointer deref
int  a4 = 17;
int *p4 = &a4;
int  b4 = *p4;   // cool

// 5: improper int assign (no deref)
int  a5 = 19;
int *p5 = &a5;
int  b5 = p5;    // ERROR
```

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`

```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 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 "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 6 weeks away. . .

CMSC216: 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
- Understanding the process abstraction of running programs, ability to create and manipulate processes
- Basic understanding of execution threads, their relation to processes, the ability to create and manipulate threads