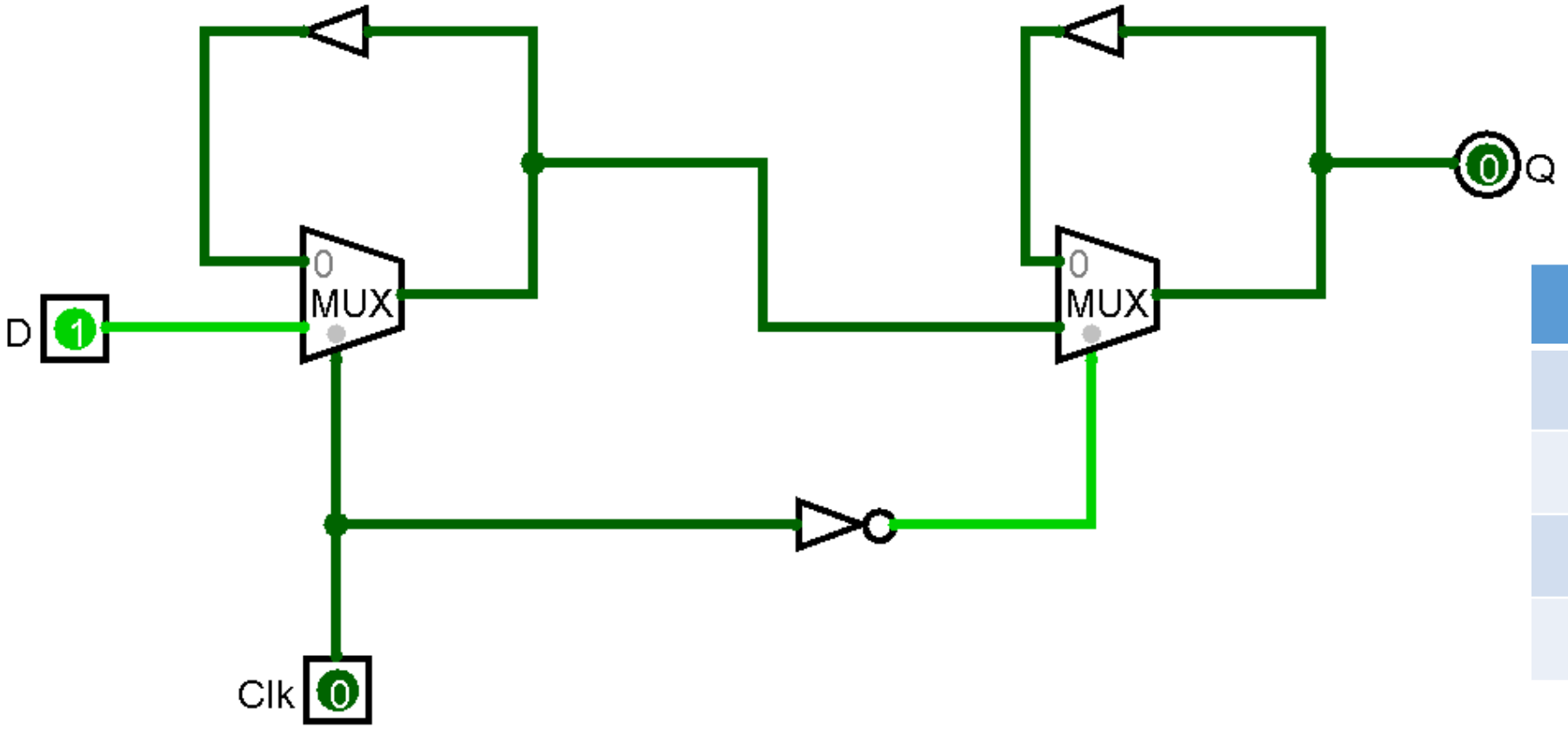
Memory and Pointers

Memory

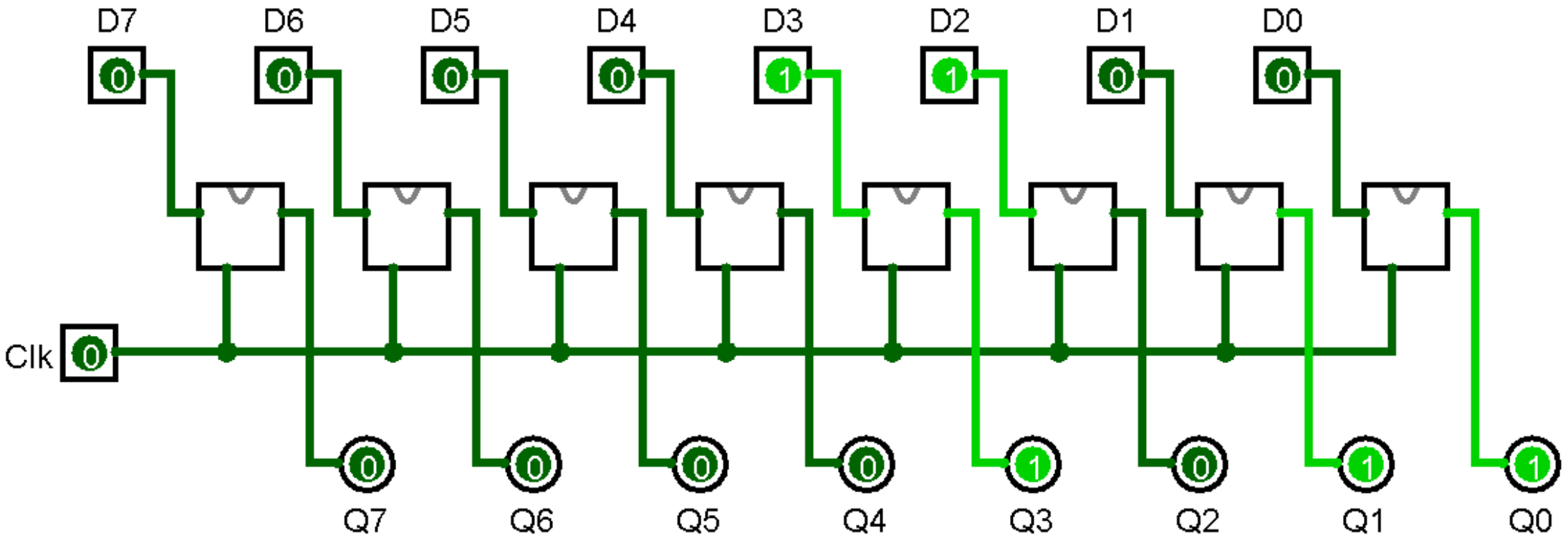
- Keeping track of something over time
- A memory is stored at one point in time
- A memory is retrieved at a later time
- Computers remember information by writing bits (1/0) to “memory”
- We retrieve information by reading bits from memory

Edge Triggered "Flip/Flop"

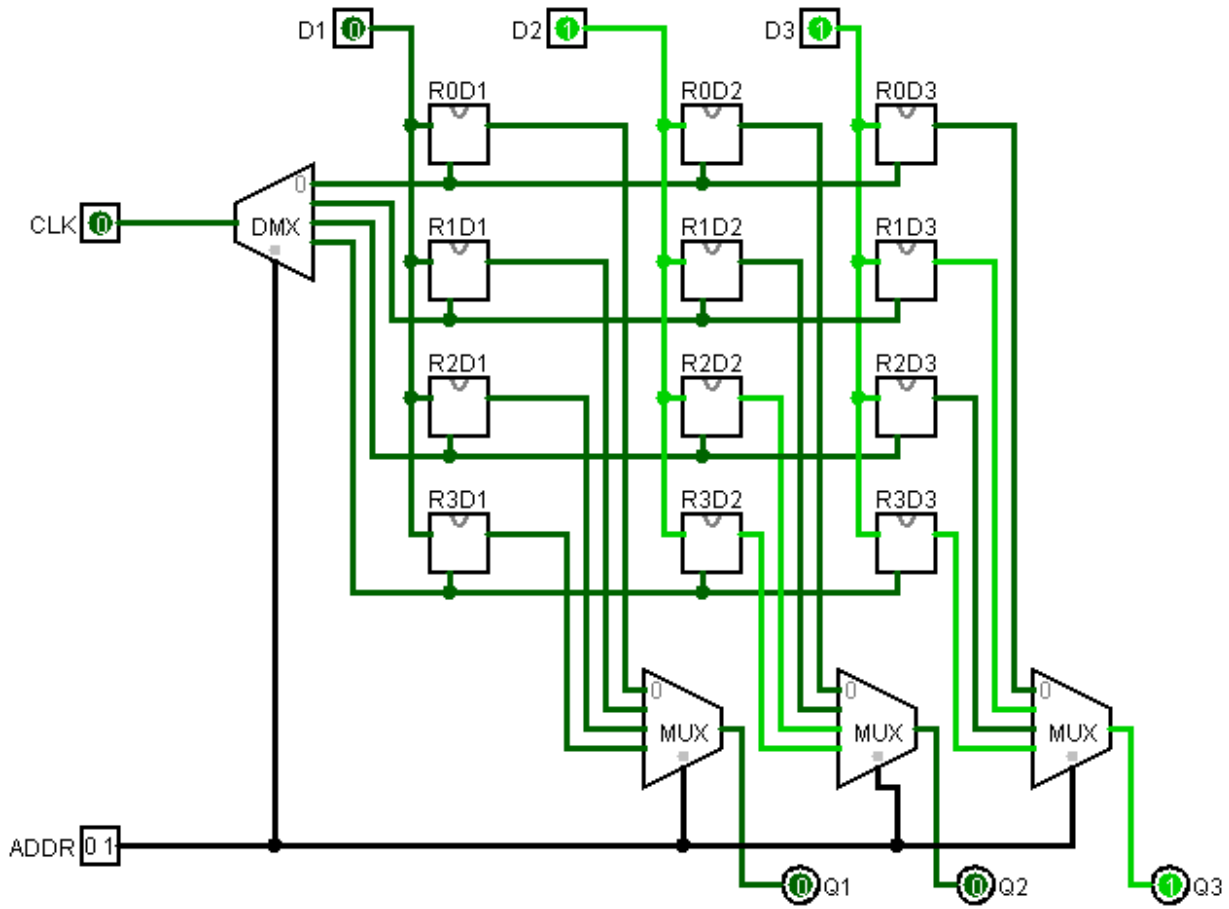


D	Clk	Q_n
X	0	Q_0
X	1	Q_0
X	0→1	Q_0
D	1→0	D

Registers – A series of Flip-Flops

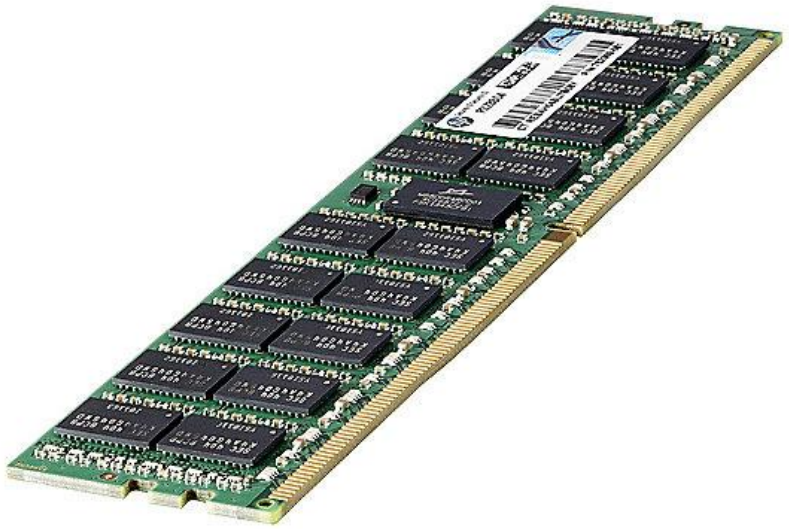


Random Access Memory (RAM)



ADDR	DATA		
00	0	0	0
01	0	0	1
10	0	1	0
11	1	1	1

Abstract view of “Main” Memory (RAM)



ADDRESS	DATA							
0xffff ffff	0	0	0	1	1	0	0	0
…								
0x0000 0002	0	0	0	1	0	1	0	0
0x0000 0001	0	0	1	1	0	1	1	0
0x0000 0000	1	1	1	0	1	1	1	1

Computer Memory Organization

- Computers read and write memory in 1 byte (8 bit) chunks
- Think of memory as a big C vector of chars:

```
char MEMORY[2_142_240_768];
```

- Like a vector, if we know the index of a byte of memory, we can either read or write to that byte:

```
MEMORY[1_684_501_289] = 'A';
printf("We stored %c\n",MEMORY[1_684_501_289];
```

Modeling "Memory"

- In computers, memory is like a RAM with 8 bit words
  - A byte is 8 bits, two hex digits, one ASCII character
  - Each byte of memory has a specific ADDRESS… the index of the byte from the beginning of memory
  - Each byte can be read or written independently
- We model this as a column with address 0 at the bottom
- For this class, we will use 64 bit addresses
  - 8 bytes, 16 hex digits, values 0-18,446,744,073,709,551,615
  - Most modern machines use 64 bit addresses
  - Slides will use 32 bit addresses so things fit
- Initial value of memory is unknown

Chap 2.1.3

Address	Value
0xFFFF FFFF	0xDE
0xFFFF FFFE	0xAD
0xFFFF FFFD	0xBE
0xFFFF FFFC	0xEF
0xFFFF FFFB	0xDE
…	
0x0000 0C07	0x00
0x0000 0C06	0x00
0x0000 0C05	0x01
0x0000 0C04	0x18
….	
0x0000 0003	0x00
0x0000 0002	0x00
0x0000 0001	0x00
0x0000 0000	0x03

Cheap Memory

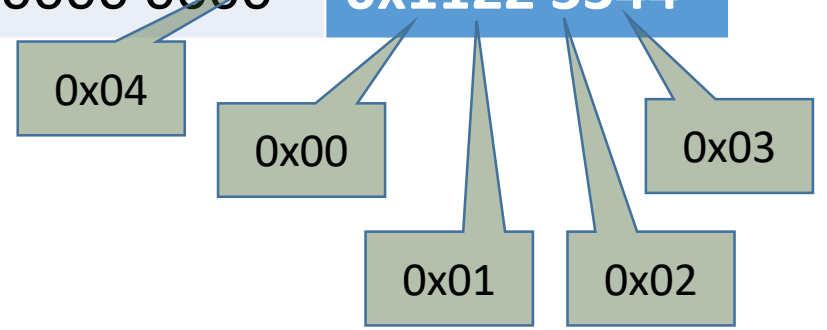
- Between Moore's Law and brilliant OS parlor tricks, "Virtual Memory" is VERY cheap!
- Memory size depends on the size of the address

Address Size	Number of Bytes addressable
2 bytes (16 bits)	2^{16} = 64K = 65,376
4 bytes (32 bits)	2^{32} = 4G = 4,284,481,536
8 bytes (64 bits)	2^{64} = 16EiB >1.8 x 10^{19}

Displaying Words of Memory

- Often we want to show multiple bytes of memory right next to each other
- For instance, an integer is 4 bytes long, and is hard to read if those bytes are not on one line
- Choose a “word size” (often 4 bytes), and show memory in words

Address	Value
0xFFFF FFFC	0xDEAD BEEF
….	
0x0000 000C	0x0000 000C
0x0000 0008	0x0000 0009
0x0000 0004	0x0000 0006
0x0000 0000	0x1122 3344

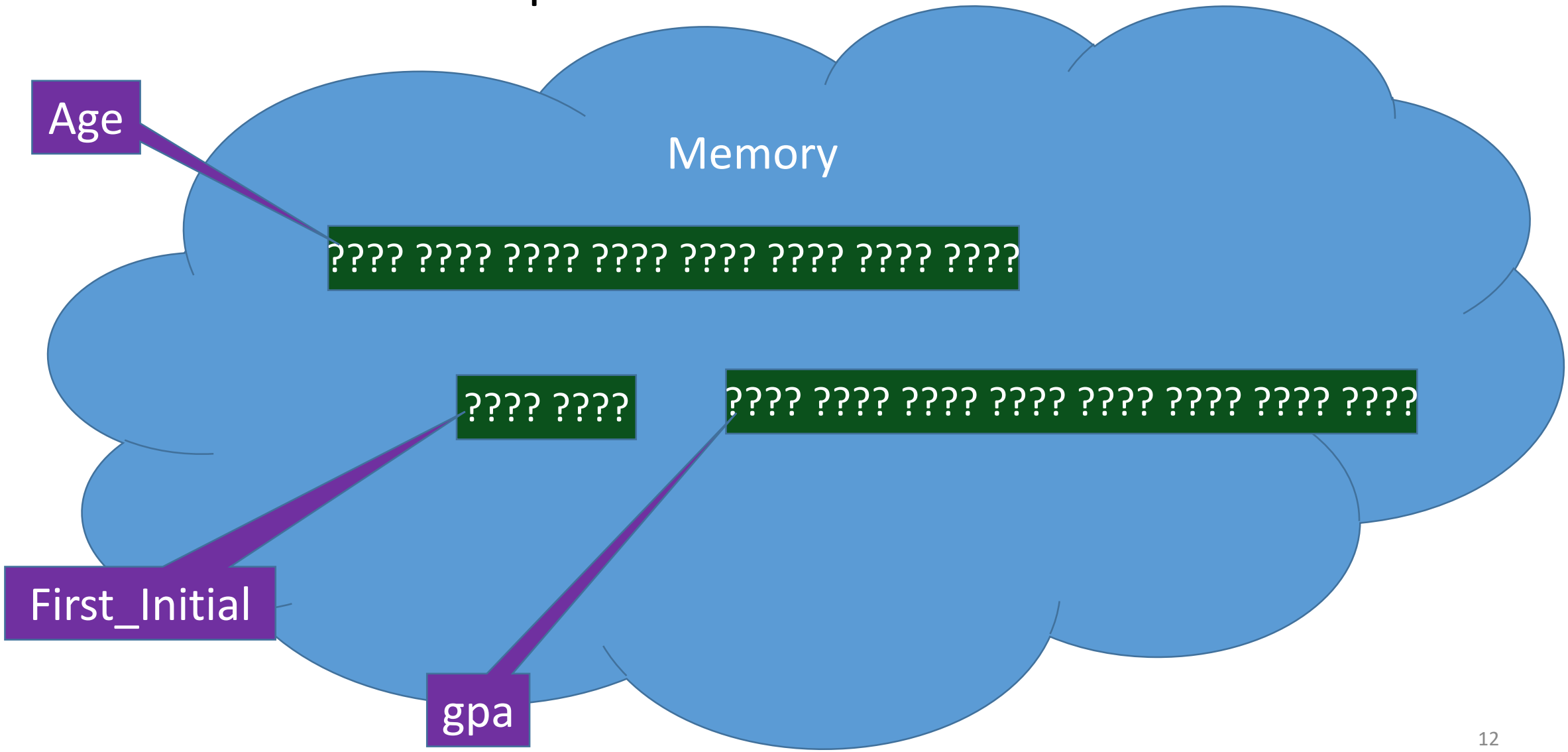


C Values

- Every "C" value resides in memory
- The "address" of a value is the location of the *beginning* of that value in memory
- The type of the value tells us how *long* to read
- Displaying 4 byte "words" in memory
- Integer @ 0x0000 0004 = 0x0000 0006 = 6_{10}
- Integer @ 0x0000 0C04 = 0x0000 0118 =280_{10}
- Display big-endian (abstractly)

Address	Value
0xFFFF FFFC	0xDEAD BEEF
0xFFFF FFF8	0xDEAD BEEF
0xFFFF FFF4	0xDEAD BEEF
0xFFFF FFF0	0xDEAD BEEF
0xFFFF FFEC	0xDEAD BEEF
…	
0x0000 0C10	0x0000 011B
0x0000 0C0C	0x0000 011A
0x0000 0C08	0x0000 0119
0x0000 0C04	0x0000 0118
….	
0x0000 000C	0x0000 000C
0x0000 0008	0x0000 0009
0x0000 0004	0x0000 0006
0x0000 0000	0x0000 0003

Variable Concept



C Variables

- The compiler reserves space in memory for each variable.
- The "address" of a value is the location of the beginning (first byte) of the value of that variable in memory

```
int height=280;
```

- We can think of the variable name as a label at a specific memory location*

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	0xFFFF FFEC	0xDEAD BEEF
	…	
	0x0000 0C10	0x0000 011B
	0x0000 0C0C	0x0000 011A
	0x0000 0C08	0x0000 0119
height	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

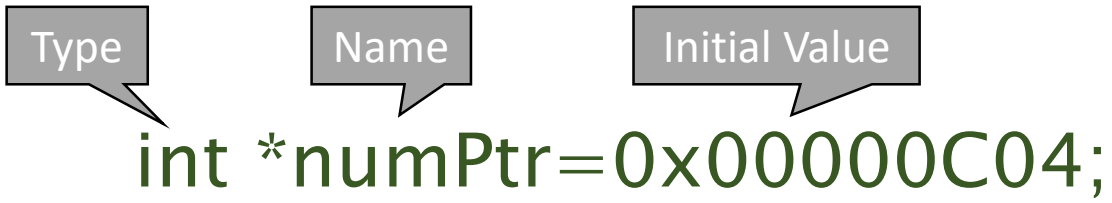
Pointers in C

- Pointers are a special class of data types
  - A variable may be declared as a pointer
- The size of a pointer is the size of an address
- The VALUE of a pointer is an address
- The TYPE of a pointer includes the type of value it is pointing to!
  - pointer to character
  - pointer to integer
  - pointer to float
  - pointer to struct date
  - …

Chap 2.1.2

Declaring Pointers

- An asterisk (*) after a data type ***in a declare statement*** means "is a pointer to"



- Type: Type of data being pointed to
- Name: Name of the pointer itself
- Value: An address
- Note: Pointers are often variables too!

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	0xFFFF FFEC	0xDEAD BEEF
	…	
	0x0000 0C10	0x0000 011B
numPtr	0x0000 0C0C	0x0000 0C04
	0x0000 0C08	0x0000 0119
height	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

"Address Of" operator

- An ampersand (&) in front of a variable means "address of" the value of that variable.

Variable

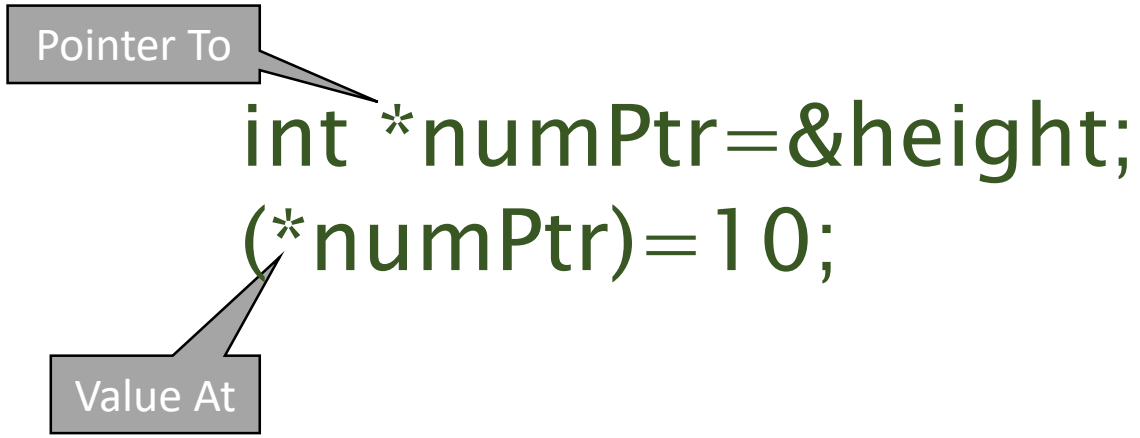
```
int *numPtr=&height;
```

- "Variable" can be any reference to memory
  - Variable name
  - Function name
  - …

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	0xFFFF FFEC	0xDEAD BEEF
	…	
	0x0000 0C10	0x0000 011B
numPtr	0x0000 0C0C	0x0000 0C04
	0x0000 0C08	0x0000 0119
height	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

"Value At" (dereference)

- An asterisk (*) in front of an expression means "value at" that expression.
- Think of *x as if it were MEMORY[x]



- Value At operator takes an address as an argument

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	0xFFFF FFEC	0xDEAD BEEF
	…	
	0x0000 0C10	0x0000 011B
numPtr	0x0000 0C0C	0x0000 0C04
	0x0000 0C08	0x0000 0119
height	0x0000 0C04	0x0000 000A
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Aliases in C

- Most languages allow only one reference to a specific piece of data
- C allows “aliasing”… multiple ways to reference a specific value

```
int x=10;
int *y=&x; // (*y) is now an alias for x
(*y)=11;
printf(“The value of x is %d\n”,x);
```

Using NULL

- "NULL" is a special address whose value is 0x0000 0000 0000 0000.
- Beginning of Memory "belongs" to the operating system
  - General programs can read at 0, but cannot write at 0
- Therefore, we use NULL to indicate "pointer to nothing"
  - Or "pointer that we haven't set yet"

```
int *p=NULL; // p is a pointer to nothing (for now)
…
p=&age; // Now p is a pointer to an integer
```

C Pitfall: “Dereferencing a Null Pointer”

```
int *p=NULL; // P is a pointer to nothing
...
if (x>0) { p=&x; }
(*p) = 5;
```

Void Pointers

```
void * myptr; // myptr is a pointer to void
```

- void * used as a “universal pointer” – a pointer to any type of data
- myptr is a pointer, but I’m not going to tell you what it points at
- Before you use (dereference) myptr, you must cast it as a pointer to something

```
printf(“myptr points to %c\n”,*(char *)myptr);
```

- Programmer must know what type of data it’s pointing at to cast correctly

Using Pointers to Pass by Reference

```
int counter=0;
void incr(int x) {
        x = x + 1;
}
incr(counter);
printf("counter=%d\n",counter);
```

counter=0

```
int counter=0;
void incrp(int *x) {
        (*x) = (*x) + 1;
}
incrp(&counter);
printf("counter=%d\n",counter);
```

counter=1

Compile vs. Dynamic (Run-Time) Memory

Compile Time Memory

- Declared in the program
- Compiler figures out how to manage this memory
  - Where it resides
  - When it is available to the program

Dynamic Memory

- Program requests chunks of memory from the “heap”, a memory pool managed by the OS
- Program manages the use of this memory
- Program must return the right to use this memory to the OS when done

Dynamic Memory

Chap 9.9

- Standard library function call to request new memory

```
#include <stdlib.h>

void * malloc(int size);
```

Number of Bytes requested

Address of space returned
NULL if no space is available
Type is pointer to nothing.

The malloc "contract"

- You are guaranteed sole use of malloc'ed memory
- Nothing outside of your program will read or write that memory
- When you are finished using that memory, you must give it back to the operating system!

```
char * buffer=(char *)malloc(300); // get 300 bytes from heap

// use buffer here

free(buffer);  // return buffer 300 bytes to the heap
```

cast to correct pointer type to avoid compiler warnings

What happens when I run out of heap?

```
int * numbers=(int *)malloc(sizeof(int)*2000000);
if (numbers==NULL) {
        printf("Ran out of heap memory!");
        exit(-1);
}
numbers[12]=16;
```

Initializing Heap Memory

- What is in malloc'ed memory?
  - Whatever was there before (Unknown values)
  - It is assumed you will write to dynamic memory before you read it
  - If this assumption is not true, use "calloc" instead of "malloc"

```
(void *) calloc(int count,int size);
```

- Allocates "count" contiguous items that are "size" bytes large
- Initializes all memory to zero (0x00)

```
float * floatVec=(float *)calloc(100,sizeof(float));
```

Managing Growing Data Structures

```
struct m * new() {
        struct m* this=(struct m*)malloc(sizeof(struct m));
        this->max=16; this->used=0;
        this->data=(int *)malloc(sizeof(int)*this->max);
        return this
}
bool add(struct m*this,int new) {
        if (this->used==this->max) {
                this->max*=2;
                this->data=(int *)realloc(this->data,sizeof(int)*this->max);
                if (this->data == NULL) return false;
        }
        this->data[this->used++]=new;
        return true;
}
```

```
struct m {
        int max;
        int used;
        int * data;
}
```

C Pitfall – Orphaned Pointers

```
int *nums = (int *)malloc(count * sizeof(int));
for(int i=0;i<count;i++) { nums[i]=foo(i); }
…
free(nums);


if (nums[1]>0) {
        printf("Wow… foo(2) was positive!\n");
        nums[12]=17;
        …
```

nums is still a valid address, but you no longer own what nums points to!

You just wrote into someone else's memory!

Core Hog Hunting

- Lazy programmers forget to free all the memory they malloc
- Eventually, you run out of heap memory
  - Especially if your program runs for days or weeks
- Kill your program and dump heap memory
- Search through the memory and try to figure out:
  - What values are in that memory
  - What code wrote those values
  - Are they still being used?

Memory Debug - valgrind

- Tracks each malloc/free and memory reference (read or write)
  - Reports on mallocs never freed
  - Reports on freed mallocs still referenced
  - Reports on references past the end of malloc'ed chunks
- Run using: valgrind *cmd parameters*
- Takes significant extra time/memory!
- Not available everywhere (not available under Cygwin)
- Free (Expensive GUI driven memory debuggers available)

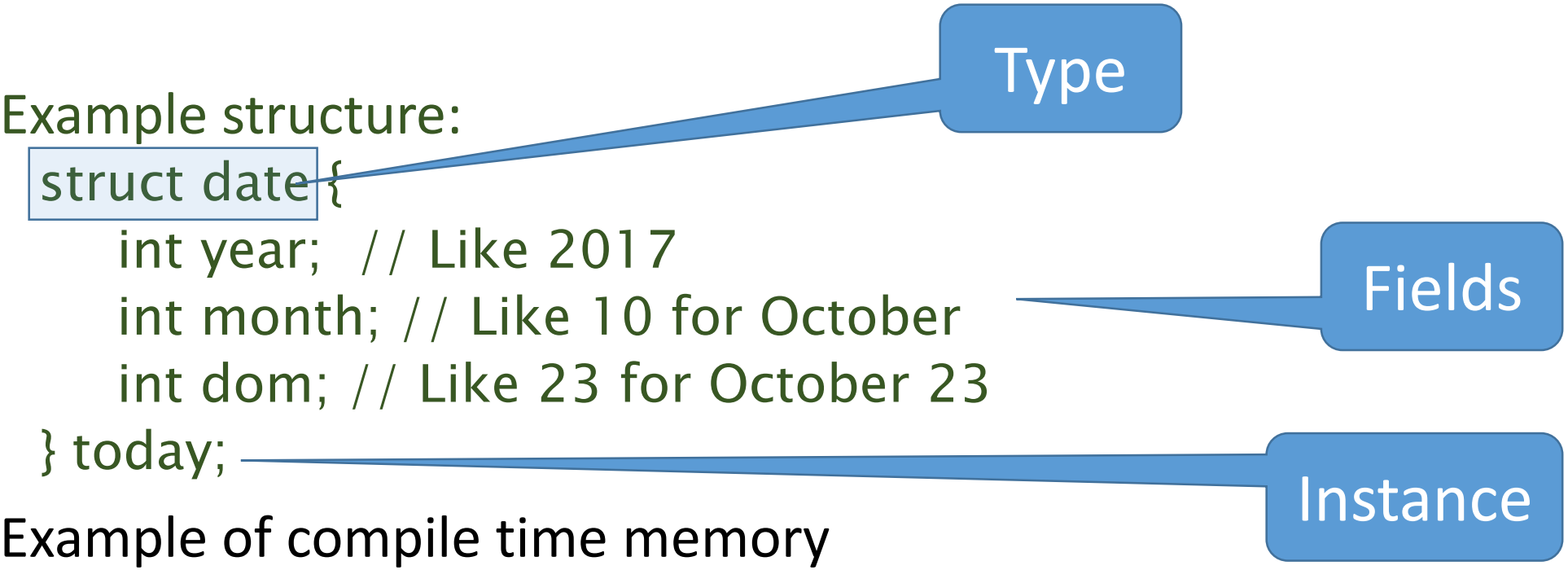
Resources

- The C Programming Language, Sections 5.1, 5.2
- Computer Systems, Section 2.1
- C-FAQ: http://c-faq.com/ptrs/index.html, http://c-faq.com/null/index.html
- Wikepedia Pointers : https://en.wikipedia.org/wiki/Pointer_(computer_programming)
- C Pointer Tutorial : http://www.tutorialspoint.com/cprogramming/c_pointers.htm

Structures and Pointers

- Example structure:

```
struct date {
    int year;  // Like 2017
    int month; // Like 10 for October
    int dom; // Like 23 for October 23
} today;
```



- Example of compile time memory
  - Reference fields with “today.year”, etc.

Structure Pointers

- Example structure pointer:
  ```
  struct date *dptr=&today;
  ```
- We could access fields with : `(*dptr).year`
- But C provides a shorthand notation: `dptr->year`

- Typical run-time structure memory allocation:
  ```
  struct date *newDate=(struct date*)malloc(sizeof(struct date));
  newDate->year=2019;
  …
  ```

Structure Pointers & Call by Reference

- Structures as arguments are call by value – entire structure is copied!
- Typically, structures are passed by reference so we can modify them

```
void tomorrow(struct date *now) {
    now->day++;
    if (now->day > daysInMonth(now->month, now->year)) …
```