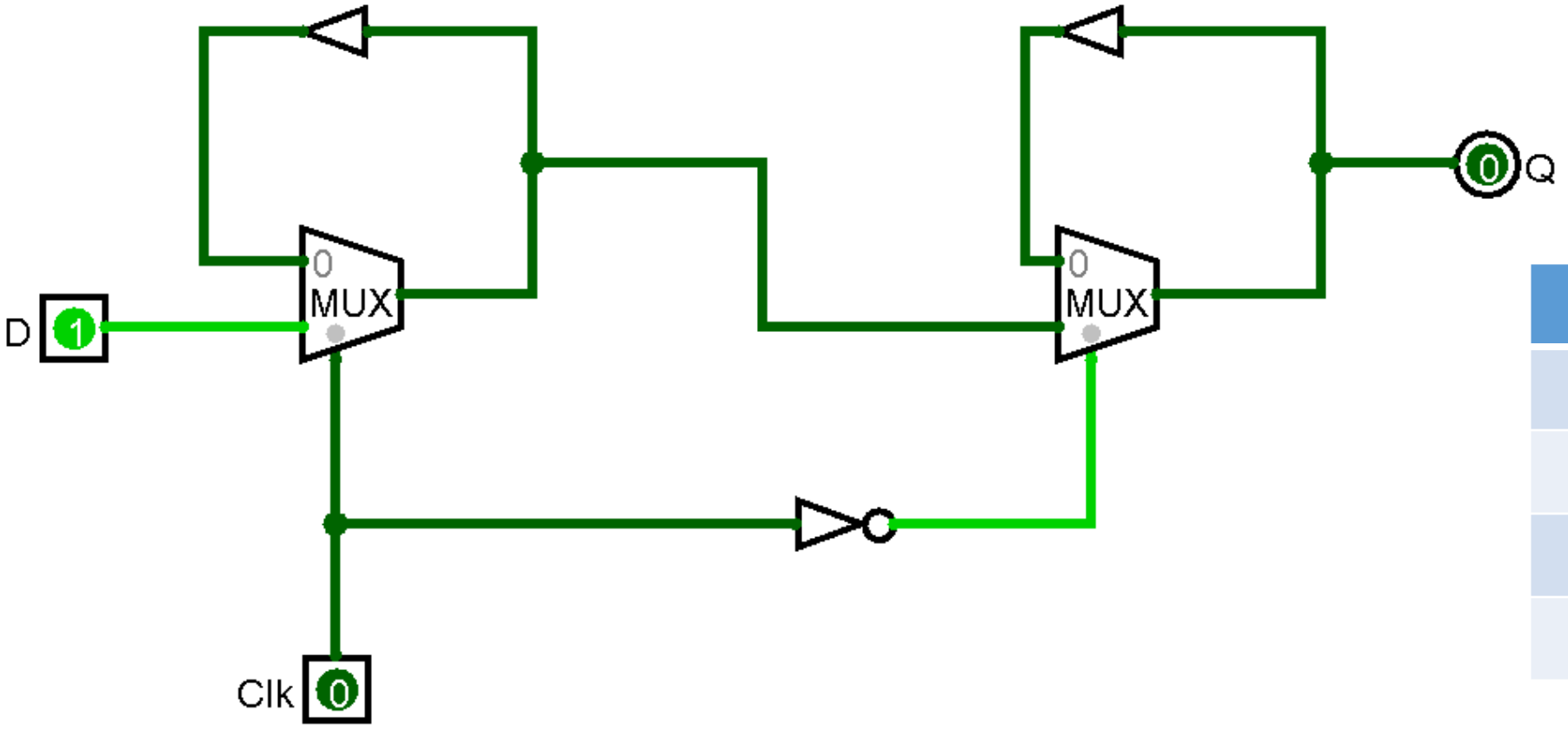
C Memory, Arrays, and Pointers

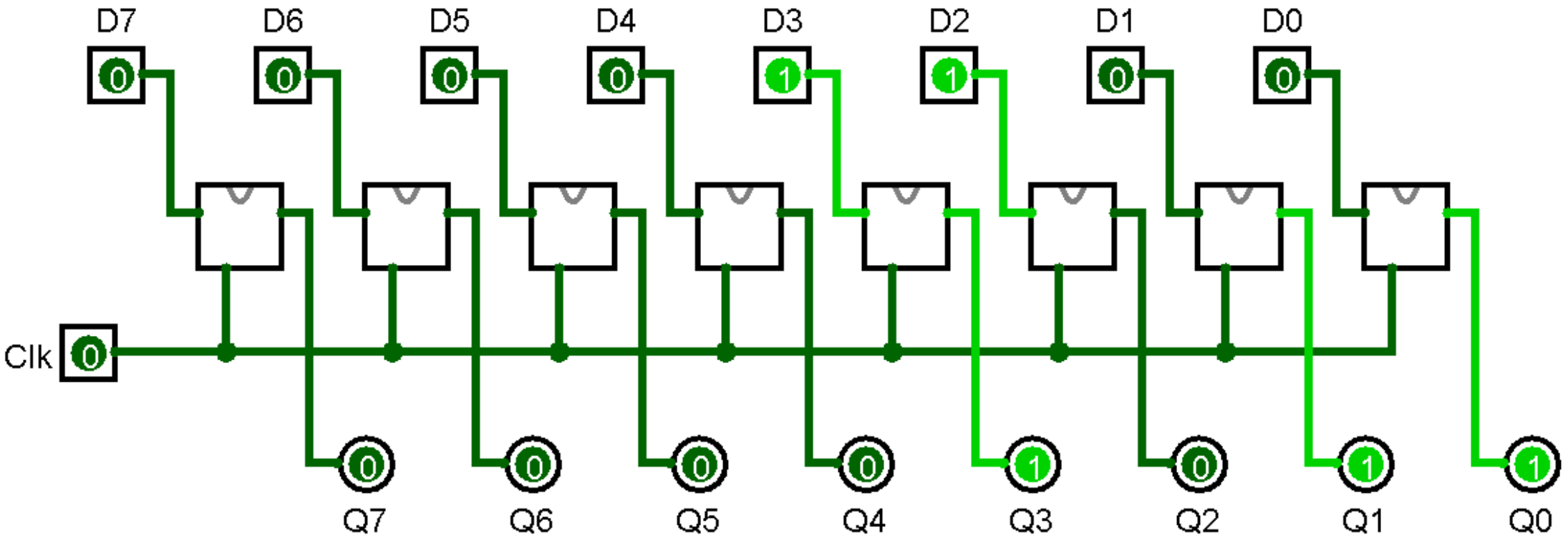
Computer Systems, Section 3.8, 6.1

Edge Triggered "Flip/Flop"

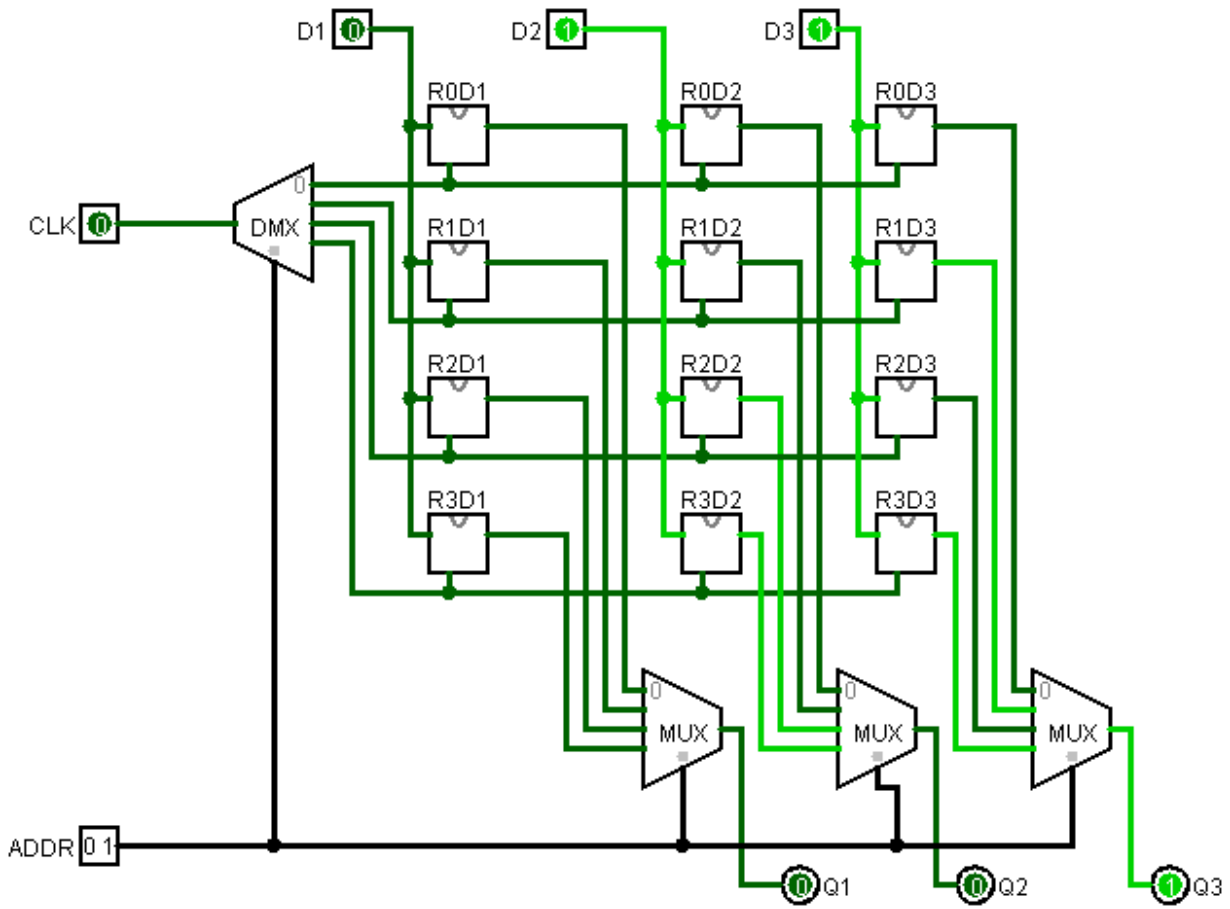


D	Clk	Q_n
X	0	Q_0
X	1	Q_0
X	0→1	Q_0
D	1→0	D

Registers


D7
D6
D5
D4
D3
D2
D1
D0
0
0
0
0
1
1
0
0
Clk
0
0
0
0
0
1
0
1
1
Q7
Q6
Q5
Q4
Q3
Q2
Q1
Q0

Random Access Memory (RAM)



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

What is "Memory"

- In computers, memory is a list of bytes
  - 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 show as a column with address 0 at the bottom
- For this class, we will use 32 bit addresses
  - 4 bytes, 8 hex digits, values 0-4G
  - Most modern machines use 64 bit addresses
- Initial value of memory is unknown

Address	Value
0xFFFF FFFF	
0xFFFF FFFE	0xDA
0xFFFF FFFD	0xED
0xFFFF FFFC	0xBE
0xFFFF FFFB	0xEF
…	
0x0000 0C08	0x00
0x0000 0C07	0x01
0x0000 0C06	0x18
0x0000 0C05	0x00
….	
0x0000 0003	0x00
0x0000 0002	0x00
0x0000 0001	0x00
0x0000 0000	0x03

Words of Memory

- A "word" is the "standard" size of data on a machine
- We will work with 32 bit words
  - 4 bytes = one integer = one float = one address
  - Old PC's had 16 bit words
  - Modern machines often have 64 bit words
- Memory as a list of words rather than a list of bytes
- Each word starts at an address divisible by 4
- Within a word, bytes go left to right*

Address	Value
0xFFFF FFFC	0xEFBE ADDE
0xFFFF FFF8	0xEFBE ADDE
0xFFFF FFF4	0xEFBE ADDE
0xFFFF FFF0	0xEFBE ADDE
0xFFFF FFEC	0xEFBE ADDE
…	
0x0000 0C10	0x001B 0100
0x0000 0C0C	0x001A 0100
0x0000 0C08	0x0019 0100
0x0000 0C04	0x0018 0100
….	
0x0000 000C	0x0C00 0000
0x0000 0008	0x0900 0000
0x0000 0004	0x0600 0000
0x0000 0000	0x0300 0000

C Values

- Every "C" value resides in memory
- The "address" of a value is the location of the *beginning* of that value in memory
- Integer @ 0x0000 0004 = 0x0600 0000
  - or 6 (little endian)
- Integer @ 0x0000 0C06 = 0x1801 0000
  - or 0x0000 0118 = 280 (little endian)

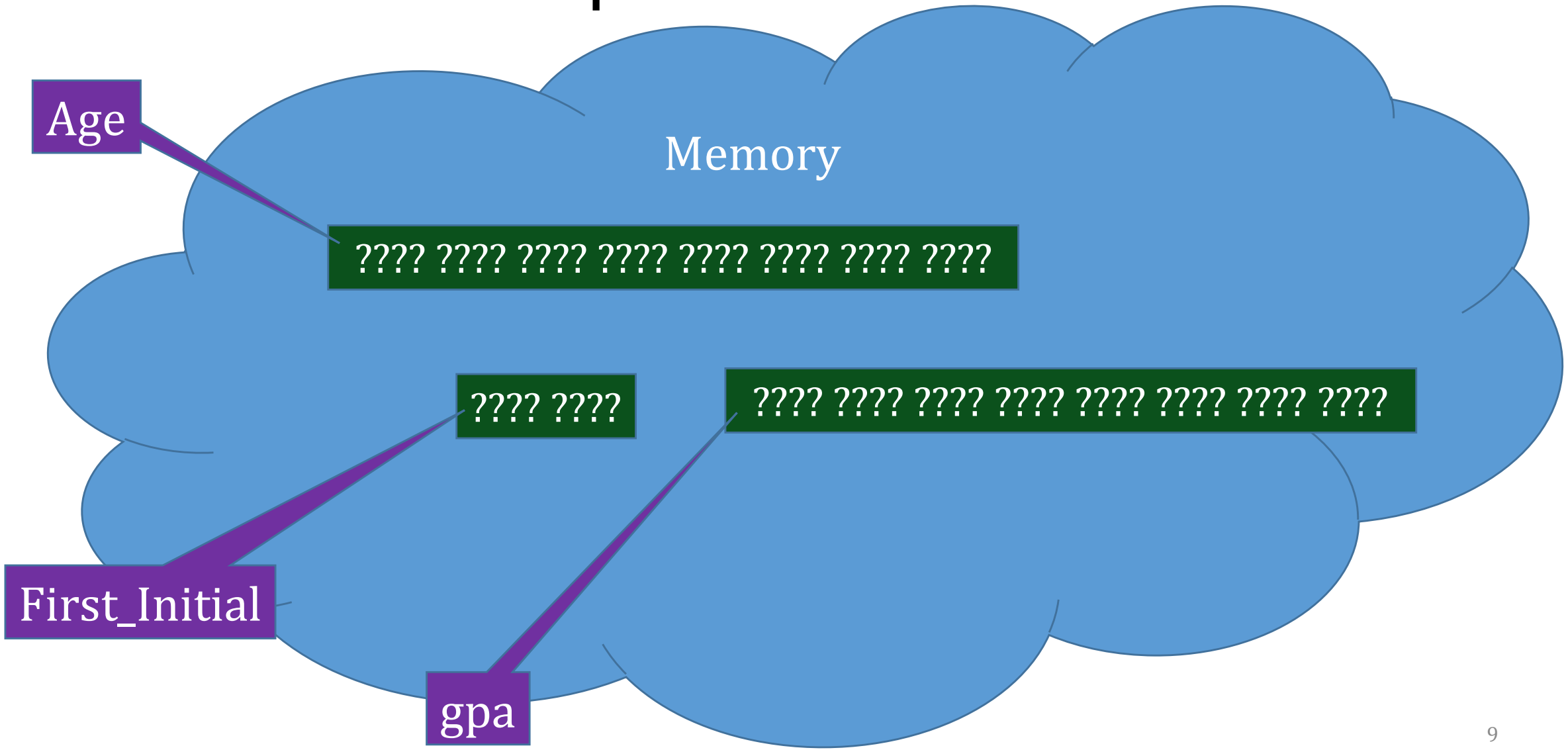
Address	Value
0xFFFF FFFC	0xEFBE ADDE
0xFFFF FFF8	0xEFBE ADDE
0xFFFF FFF4	0xEFBE ADDE
0xFFFF FFF0	0xEFBE ADDE
0xFFFF FFEC	0xEFBE ADDE
…	
0x0000 0C10	0x1B01 0000
0x0000 0C0C	0x1A01 0000
0x0000 0C08	0x1901 0000
0x0000 0C04	0x1801 0000
….	
0x0000 000C	0x0C00 0000
0x0000 0008	0x0900 0000
0x0000 0004	0x0600 0000
0x0000 0000	0x0300 0000

Little vs. Big Endian

- When we examine memory, if we look at individual bytes, we see the effect of little-endian
- If we look at words, the libraries used to print those words switch ends for us, so we "see" big-endian, even when the data itself is little endian.
- From now on, we will show words big-endian (bytes right to left)*

Address	Byte Value	Word Value
0xFFFF FFFC	0xEFBE ADDE	0xDEAD BEEF
0xFFFF FFF8	0xEFBE ADDE	0xDEAD BEEF
0xFFFF FFF4	0xEFBE ADDE	0xDEAD BEEF
0xFFFF FFF0	0xEFBE ADDE	0xDEAD BEEF
0xFFFF FFEC	0xEFBE ADDE	0xDEAD BEEF
…		
0x0000 0C10	0x1B01 0000	0x0000 011B
0x0000 0C0C	0x1A01 0000	0x0000 011A
0x0000 0C08	0x1901 0000	0x0000 0119
0x0000 0C04	0x1801 0000	0x0000 0118
….		
0x0000 000C	0x0C00 0000	0x0000 000C
0x0000 0008	0x0900 0000	0x0000 0009
0x0000 0004	0x0600 0000	0x0000 0006
0x0000 0000	0x0300 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 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

C Pointers

- Each value is at a specific memory location.

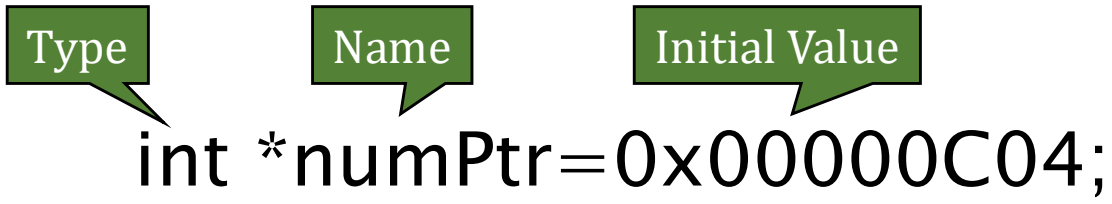
```
int height=280;
```

- We can use the address of the value in memory (e.g. 0x0000 0C04) as an alternate label
- Called a "pointer" to a value
- Pointers are 32 bits (8 hex digits)

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

Declaring Pointers

- An asterisk (*) in front of 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: The address of the data

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 an expression means "address of" that expression.

Expression

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

- Expression may 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" operator

- An asterisk (*) in front of an expression means "value at" that expression.

Pointer To

```
int *numPtr=&height;
(*numPtr)=10;
```

Value At

- Value At operator takes an address as an argument
- Value at can be used to read or write data

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

Type Checking with Pointers

- int *x; // x is a pointer to a signed integer
- int **y; // y is a pointer to a pointer to a signed integer
- &z – Type is: pointer to <type of z>
- (*myptr) – Type is: type which myptr is pointing to
  e.g. (*numPtr)=“string here”;

“Unable to assign ‘char *’ to int”

- Special pointer type: void * - pointer to a unspecified type
  - void * pointers can be cast to pointers to any type!
  - Used as “universal” pointers

C Gotcha: "Dereferencing a Null Pointer"

- "NULL" in C is a macro defined to 0x0000 0000
  - Note: NULL is a 4 byte word "0", or address 0
- NULL address is used to indicate that this pointer is not yet set
  - 0 is an address which "belongs" to the operating system
  - Programs can read at 0, but cannot write at 0

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

Segmentation Violation when x<=0

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);
```

C Arrays

- List of contiguous values in memory
- Array Declaration:

- Type: Type of each element
- Name: Identifier for the entire array
- Count: Number of elements in the list

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	…	
vec[4]	0x0000 0C14	0x1111 1111
vec[3]	0x0000 0C10	0x1111 1111
vec[2]	0x0000 0C0C	0x1111 1111
vec[1]	0x0000 0C08	0x1111 1111
vec[0]	0x0000 0C04	0x1111 1111
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Array Element Access

- Square Bracket Operator
  - Argument: Index from 0 to (Count-1)
- Example:

```
for(i=0;i<5;i++) {
        vec[i]=280+i; }
```

- Can be used to read or write an array element

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	…	
vec[4]	0x0000 0C14	0x0000 011C
vec[3]	0x0000 0C10	0x0000 011B
vec[2]	0x0000 0C0C	0x0000 011A
vec[1]	0x0000 0C08	0x0000 0119
vec[0]	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Array Name

- In C, by convention, the array name is the address of the first element of the array

  vec=&(vec[0])

- Therefore, the following holds:

$$\&(vec[i]) == (char\ *)vec + sizeof(vec[0]) * i$$

Pointer Arithmetic

- You can do math (+-*/%) with pointers, but…
- A **"unit"** in pointer arithmetic is the size of the data type pointed to by the pointer

```
int *x; int vec[5];
for (x=vec; x<=&vec[5]; x++) (*x)=3;
```

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	…	
x	0x0000 0D10	0x0000 0C18
	…	
vec[4]	0x0000 0C14	0x0000 0003
vec[3]	0x0000 0C10	0x0000 0003
vec[2]	0x0000 0C0C	0x0000 0003
vec[1]	0x0000 0C08	0x0000 0003
vec[0]	0x0000 0C04	0x0000 0003
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Abstraction

An array is an indexable list of data items

```
char buffer[200]; // buffer is a list of 200 characters
buffer[0]='H'; // set the first item in buffer to 'H'

int length[3]; // length is a list of 3 integers
length[0]=12; length[1]=12; length[2]=12;
```

Leaky Abstraction

- An array can be treated as an indexable list of data items but…
- An array is a contiguous list of data items in memory or …
- A contiguous list of data items in memory is an array

```
int array[10];
```

```
array == &(array[0]) or (*array)=array[0]
array[i] ==*(array+i);
```

Pointer / Array Ambiguity

- In C, we can treat pointers like arrays, and arrays like pointers

Using array notation

```
int strlen(char str[]) {
        int i=0;
        while(str[i]!=0) {
                i++;
        }
        return i;
}
```

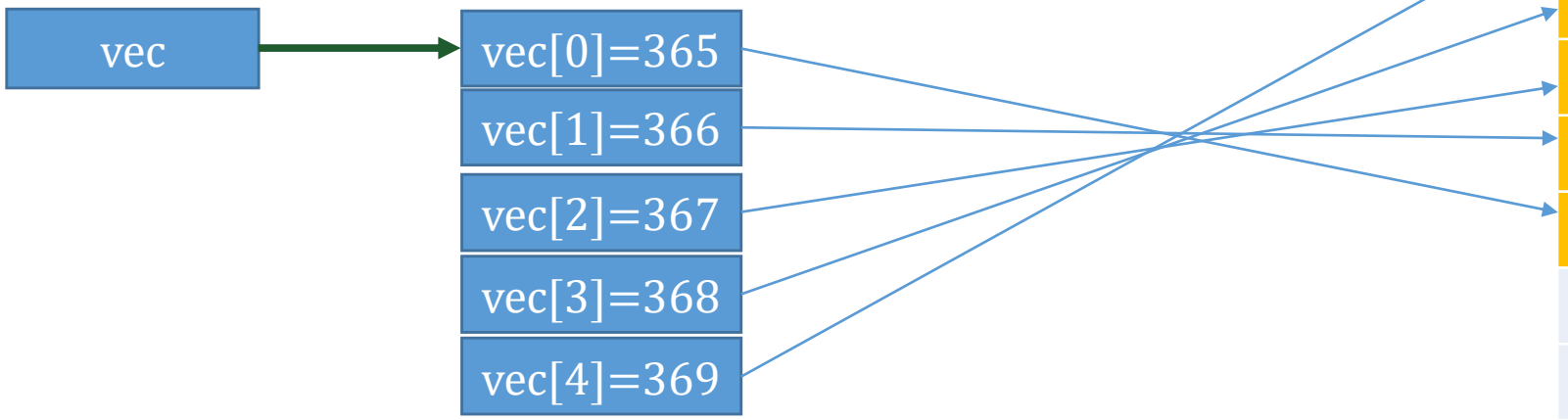
Using pointer notation

```
int strlen(char *str) {
        int i=0;
        while((*str)!=0) {
                i++; str++;
        }
        return i;
}
```

Inverted Arrays

- Standard representation of arrays in CS is top to bottom.

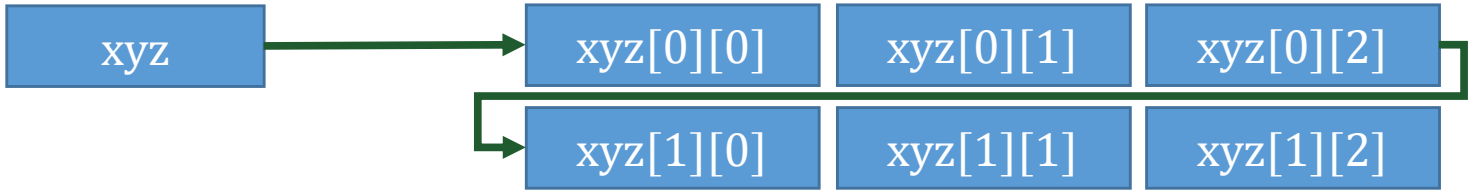
int vec[5]={365,366,367,368,369};



Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	…	
vec[4]	0x0000 0C14	0x0000 011C
vec[3]	0x0000 0C10	0x0000 011B
vec[2]	0x0000 0C0C	0x0000 011A
vec[1]	0x0000 0C08	0x0000 0119
vec[0]	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Multi-Dimensional Arrays

```
int xyz[2][3]; // 2 rows/3 cols
```

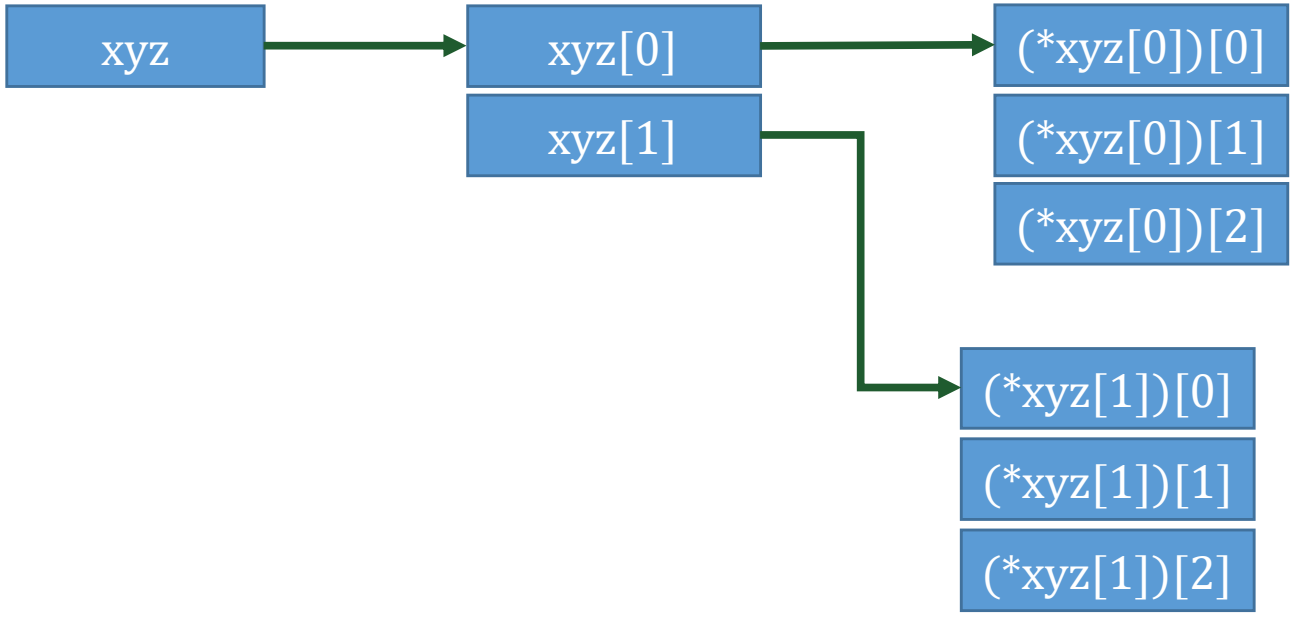


- Think of multi-dimensional array indexes like an odometer… rightmost digit first

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	…	
xyz[1][2]	0x0000 0C18	0x0000 011D
xyz[1][1]	0x0000 0C14	0x0000 011C
xyz[1][0]	0x0000 0C10	0x0000 011B
xyz[0][2]	0x0000 0C0C	0x0000 011A
xyz[0][1]	0x0000 0C08	0x0000 0119
xyz[0][0]	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Arrays of Pointers

int *xyz[2]; // 2 pointers to lists



Label	Address	Value
	…	
xyz[1]	0x0000 CA0c	0x0000 C708
xyz[0]	0x0000 CA08	0x0000 C000
	…	
xyz[1][2]	0x0000 C710	0x0000 0005
xyz[1][1]	0x0000 C70C	0x0000 0004
xyz[1][0]	0x0000 C708	0x0000 0003
	…	
xyz[0][2]	0x0000 C008	0x0000 0002
xyz[0][1]	0x0000 C004	0x0000 0001
xyz[0][0]	0x0000 C000	0x0000 0000
	…	

Difference between Arrays and Pointers

Array

- Fixed, pre-defined length
- Space reserved by compiler
- All 2D rows equal length

Pointers to List

- No pre-defined length
- No space reserved
- 2D rows may be varying length

Arrays with Undefined Sizes

- C allows the first dimension of an array to be of unspecified size
- e.g. float vec[]; // vector of unspecified length
- e.g. int m[][3]={0,1,2,3,4,5,6,7,8};
  - m is a matrix of an unspecified number of rows
  - Each row in m has 3 columns
  - Space is reserved for 9 values (based on initializer) or 3 rows

- Why does C require all but the first dimension to be specified?

Dealing with Undefined lengths

- Implicit array length
  int triArea(int *sides); // sides points to three integers

- Explicit independent length
  int polyArea(int n, int sides[]); // sides points to n integers

- "Guard" values
  int polyArea(int * sides) { // sides points to list of positive integers
    for(; (*sides)>0; sides++) { // Deal with next side…

Character Strings

- In C, there is no “string” type
- Text consists of an array of characters

```
char nextLine[80];
char bgColor[]=“magenta”; // bgColor has 8 values
```

- By default “Guard” is a “null terminator” = 0x00
  - Note: NOT “NULL”

Working with Strings

```
char *name="Thomas";
printf("name[0] is at %p\n",&name[0]);
printf("name value is %x\n",name);
printf("name string is %s\n",name);
```

```
name[0] is at 0x23cb10
name value is 23cb10
name string is Thomas
```