Pointers

CSCI 237: Computer Organization
5th Lecture, Feb 19, 2025

Jeannie Albrecht

Administrative Details

- Six puzzles due today/tomorrow
- All puzzles due next week
- First Glow HW due Friday
- Questions?

Last time: Integers

- Representing information as bits
- Bit-level manipulations
- Integers (Ch 2.2)
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Today: Pointers

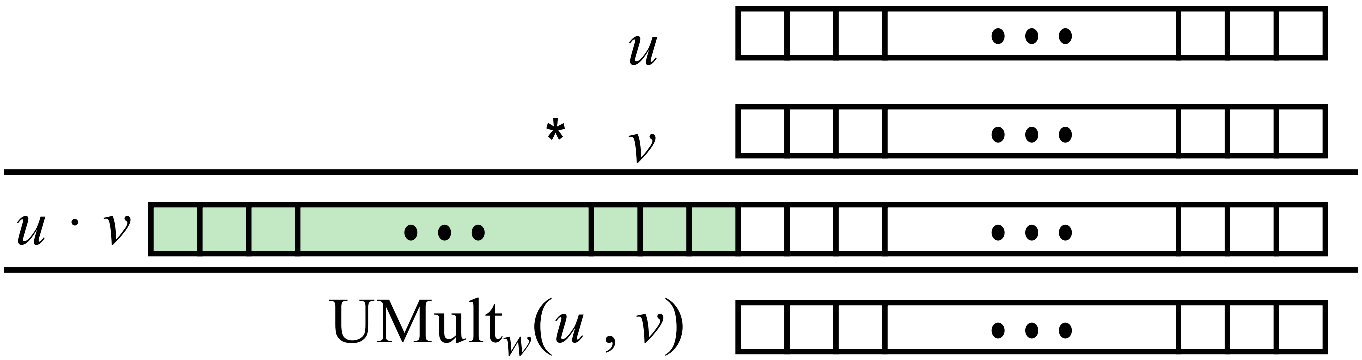
- Integers (Ch 2.2)
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, **shifting**
  - **Summary**
- **Representations in memory, pointers, strings in C**
- Background: Fractional binary numbers
- IEEE FP standard (normalized and denormalized values)
- Example and properties
- Floating point in C
- Summary

Recap: Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order *w* bits
- Implements Modular Arithmetic

$\mathrm{UMult}_w(u, v) = u * v \bmod 2^w$

```
               1110 1001         E9         233
*              1101 0101     *   D5     *   213
  1100 0001 1101 1101         C1DD        49629
               1101 1101           DD         221
```

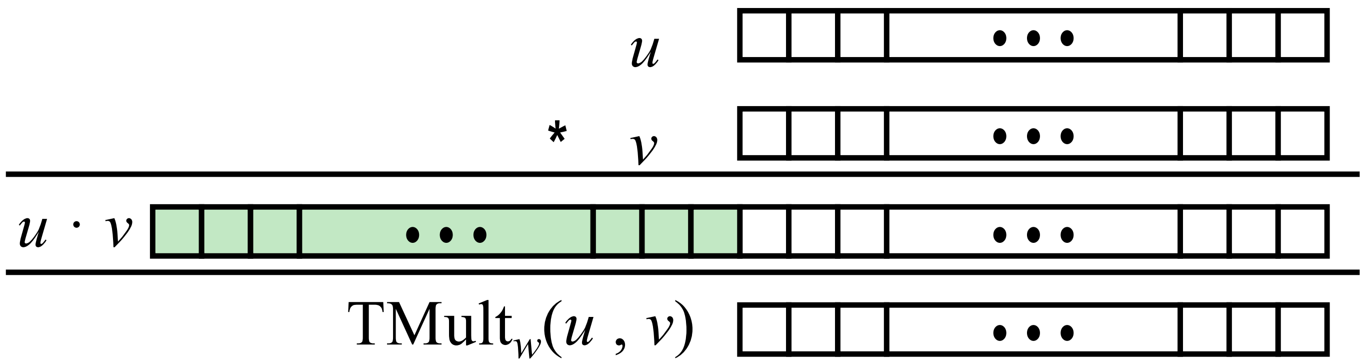
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Recap: Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order *w* bits
  - *Some of which are different for signed vs. unsigned multiplication*
  - Lower bits are the same (as unsigned multiplication)

```
  1111 1111 1110 1001       E9        -23
 *1111 1111 1101 0101   *   D5   *    -43
 --------------------     ----     ------
  0000 0011 1101 1101     03DD        989
 --------------------     ----     ------
            1101 1101       DD        -35
```

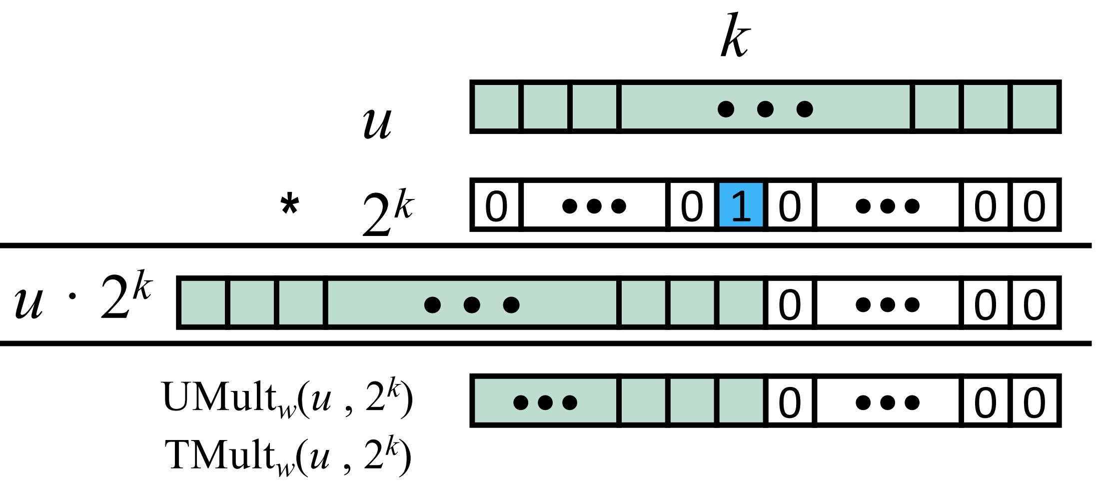
Power-of-2 Multiply with Shift

- Operation
  - **`u << k`** gives **`u * 2`**^{k}
  - Both signed and unsigned

Operands: *w* bits

True Product: *w*+*k* bits

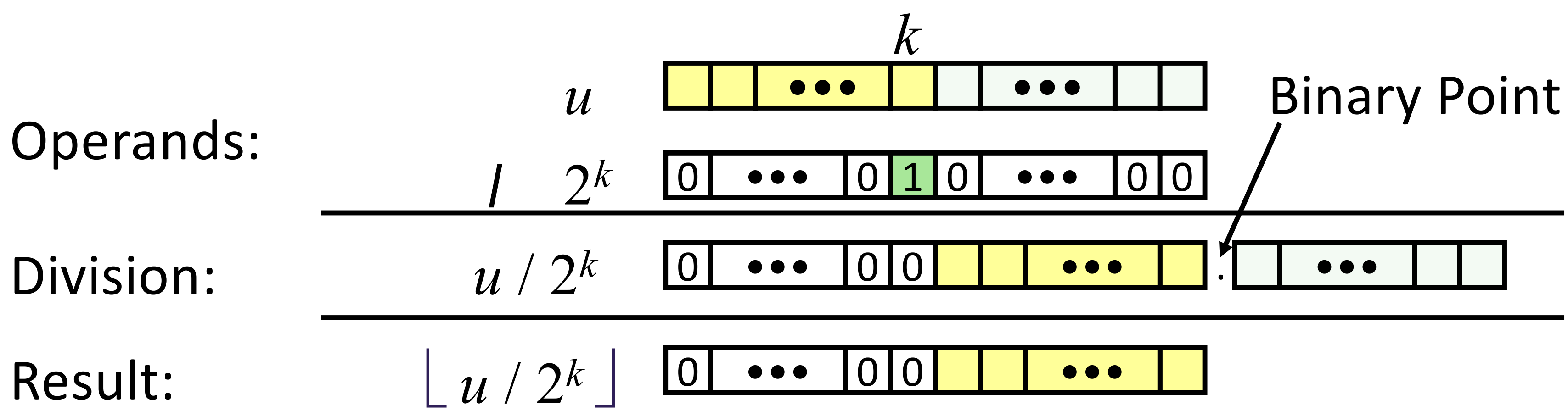
Discard *k* bits: *w* bits



- Examples
  - **`u << 3`** **`==`** **`u * 8`**
  - **`(u << 5) - (u << 3)`** **`==`** **`u * 24`**
  - Most machines shift and add much faster than multiply
    - Rewrite expressions to use shifts instead of multiply
    - Compiler tries to generate this code automatically

Unsigned Power-of-2 Divide with Shift

- Operation
  - `u >> k` gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift

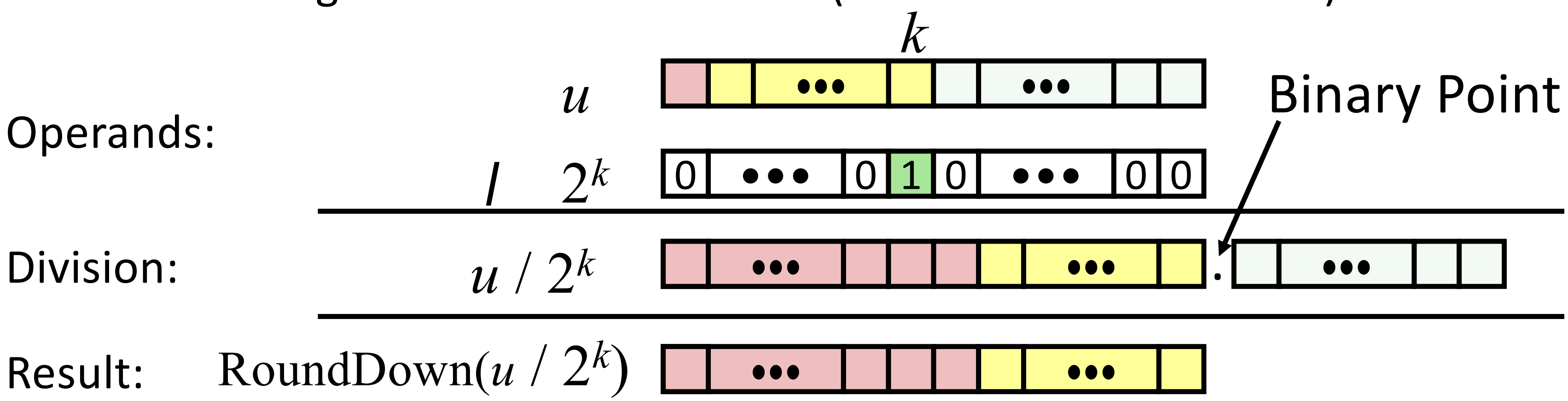


	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Signed Power-of-2 Divide with Shift

- Operation
  - **u >> k** gives $\lfloor u\ /\ 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0` (should round toward 0!)

We want::
$\lfloor u / 2^k \rfloor$ **if u > 0, but**
$\lceil u / 2^k \rceil$ **if u < 0**



	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Signed Power-of-2 Divide with Shift

- Desired operation behavior
  - Want $\lfloor$ **`u / 2`**^{k} $\rfloor$ for positive quotients (Round Toward 0)
  - Want $\lceil$ **`u / 2`**^{k} $\rceil$ for negative quotients (Round Toward 0)
  - Compute as $\lfloor$ **`(u+2`**^{k}**`-1) / 2`**^{k} $\rfloor$
    - In C: **`(u + (1<<k)-1) >> k`**
    - **`2`**^{k}**`-1`** Biases dividend toward 0

Add 2^k-1 to u as a "bias":
If dividing by 2^k, add bitvector with rightmost k bits set to 1

Understanding the Bias:

Divide by Power of 2 (k = 2, divide by 4)

	-16/4 = -4	-17/4 = -4.25	-18/4 = -4.5	-19/4 = -4.75
	1100 00	1011 11	1011 10	1011 01
Divide by Shifting 2	111100 = -4	111011 = -5	111011 = -5	111011 = -5
Add bias	1100 00 + 0000 11 ----------------- 1100 11	1011 11 + 0000 11 ----------------- 1100 10	1011 10 + 0000 11 ----------------- 1100 01	1011 01 + 0000 11 ----------------- 1100 00
Divide by Shifting 2	111100 -4	111100 -4	111100 -4	111100 -4

Negation: Complement & Increment

- Negate through complement and increase

  `~x + 1 == -x`

- Example
  - Observation: `~x + x == 1111…111 == -1`

Flip the bits and add 1
(most of the time!)

	x	1 0 0 1 1 1 0 1
+	~x	0 1 1 0 0 0 1 0
	-1	1 1 1 1 1 1 1 1

x = 15213

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`
`~x`	-15214	`C4 92`	`11000100 10010010`
`~x+1`	-15213	`C4 93`	`11000100 10010011`

Complement & Increment Examples

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

x = TMin

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
~x	32767	7F FF	01111111 11111111
~x+1	-32768	80 00	10000000 00000000

Canonical counter example to ~x + 1 == -x

Summary: Basic Arithmetic Rules

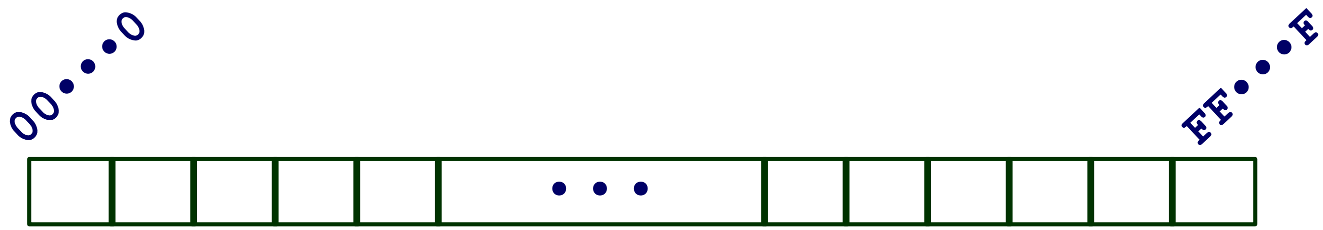
- Addition:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper signed range)

Today: Pointers

- Integers (Ch 2.2)
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- **Representations in memory, pointers, strings in C**
- Background: Fractional binary numbers
- IEEE FP standard (normalized and denormalized values)
- Example and properties
- Floating point in C
- Summary

Byte-Oriented Memory Organization



- Programs refer to data by their memory address
  - Conceptually, envision memory as a very large array of bytes
    - (In reality, it's not, but we can think of it that way…)
  - An address is like an index into that array
    - Each variable is assigned an address
    - "Type" of a variable tells us how to interpret bytes at the variable's associated address
- Note: system provides private *address spaces* to each *process*
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others (more on this later)

Machine *Words*

- Any given computer has a "word size" (aka *w* in previous slides)
  - Word size is the size of an address (aka a *pointer*)

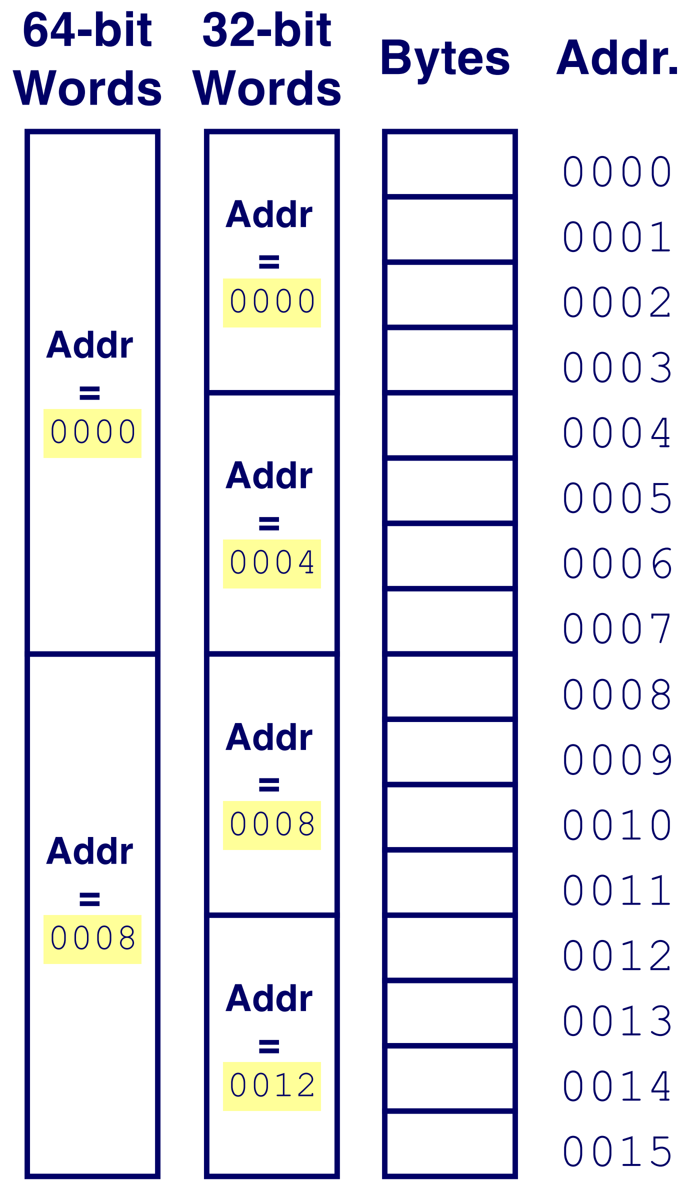
- Historically, most machines used 32-bit words (4 bytes)
  - This limited addresses (memory size) to 4GB (2^{32} bytes)

- Now, most machines have 64-bit word size
  - Potentially, could have 18 EB (exabytes) of addressable memory!
  - That's 18.4 X 10^{18}

- Machines still support multiple data formats
  - Primitive types are always an integral number of bytes
  - Fractions (e.g., `char`, `short`) or multiples (e.g., `long int`) of word size

Word-Oriented Memory Organization

- Addresses specify byte locations
  - Address “points to” first byte in word
  - Addresses of successive words always differ by 4 (32-bit) or 8 (64-bit)

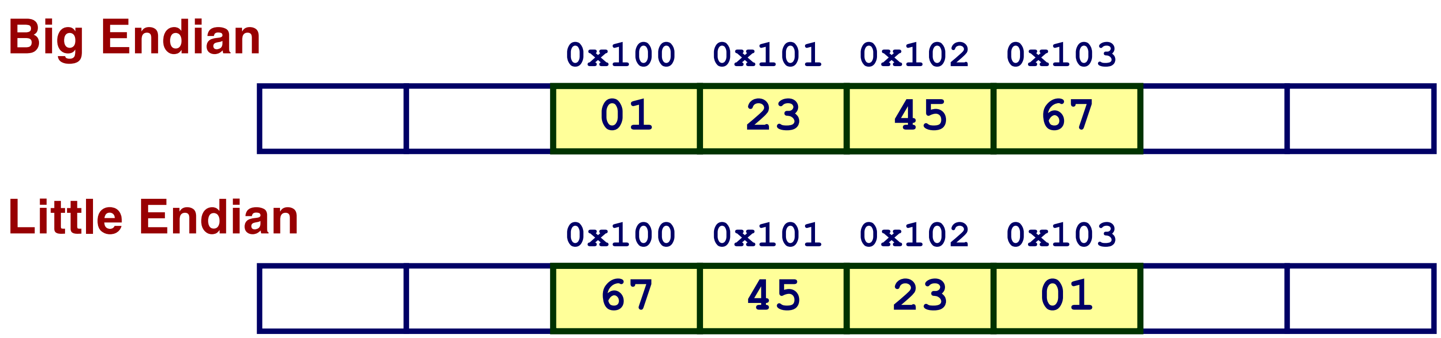


Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

Byte Ordering Example

- Example
  - Let variable `x` have 4-byte value of 0x01234567
  - The address given by `&x` is 0x100
    - & is the “address operator” in C. Gives you the mem address of x



Big Endian: Least significant byte has highest address
Little Endian: Least significant byte has lowest address

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
`pointer`	4	8	8

Pointers

- Pointers store a memory address
- A variable’s memory address can be obtained with `&`
  - `int val = 3;`
  - `int *ptr = &val;        // Same as int *ptr; ptr = &val;`
- The pointer variable stores a value that is the address
  - `ptr == NULL`
- To access the value pointed to, need to use the * operator
  - `int num = *ptr;`
  - `*ptr = 7;`
- Modifying the pointer variable without the * operator changes what is pointed to
  - `ptr = #`
  - `*ptr = 10;`

```
int val = 4;

int *ptr = NULL;

ptr = &val;

*ptr = 7;
```

addr = 0x1000

7

addr = 0x1004

0x1000

Pointer Practice

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x = 3, y = 10;
    int *ptr = NULL;

    ptr = &y;
    *ptr = 7;
    y++;
    printf("x %d y %d ptr %d\n", x, y, *ptr);

    ptr = &x;
    *ptr = *ptr + 1;
    printf("x %d y %d ptr %d\n", x, y, *ptr);

    ptr = NULL;
    // Why doesn't the next line of code work?
    printf("x %d y %d ptr %d\n", x, y, *ptr);
    return 0;
}
```

x 3 y 8 ptr 8

x 4 y 8 ptr 4

ptr == NULL

Representing Arrays In Memory

- Each array element is the size of the specified data type
- Array elements laid out in memory in *contiguous* memory locations
- Array name also refers to the *address of* the first element in the array
  - `arr == &arr[0]`
- Any operation done with an array can also be done with pointers (aka pointer arithmetic)
  - `int *ptr = arr;`
  - `ptr == &arr[0]`
  - `(ptr+1) == &arr[1]`

```
int arr[6];
```

arr	
31	arr[0]
38	arr[1]
32	arr[2]
31	arr[3]
33	arr[4]
00	arr[5]

```c
int main(int argc, char *argv[]) {
  int arr[4];

  arr[0] = 0;
  arr[1] = 1;
  arr[2] = 2;
  arr[3] = 3;

  int *ptr = arr;
  // Above equivalent to
  // int *ptr;
  // ptr = arr;

  printf("Addresses: \n");
  printf("\t arr     \t %llx \n", (unsigned long long) arr);
  printf("\t &arr[0] \t %llx \n", (unsigned long long)(&arr[0]));
  printf("\t ptr     \t %llx \n", (unsigned long long) ptr);
  printf("\t &ptr    \t %llx \n", (unsigned long long) &ptr);

  return 0;
}
```

Addresses:

```
arr         7ffee62258a0
&arr[0]     7ffee62258a0
ptr         7ffee62258a0
&ptr        7ffee6225888
```

Why use pointers?

- Modular programming
  - Allows you to pass the memory address of a local variable to another function
    - Allows other function to read and update variable so that modifications are reflected in function where variable created
  - Avoids need to create a copy of the variable contents (so less memory is used)
- Enables single name for collection of elements/memory addresses (i.e., array)
  - Only need one pointer/array name to access all of the elements of an array

Representing Strings

```
char S[6] = "18213";
```

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be **null-terminated**
    - Final character = 0
- Compatibility
  - Byte ordering not an issue

ASCII

31
38
32
31
33
00

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```

arr: →

D	o	n	u	t	s	!			

ptr:

```
arr Donuts!
```

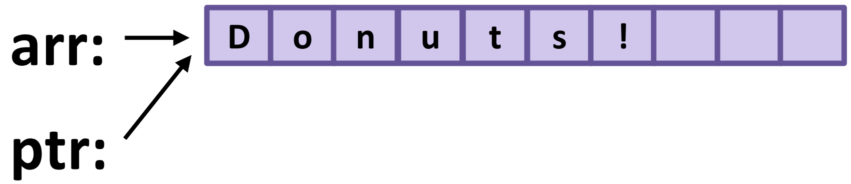
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!
```

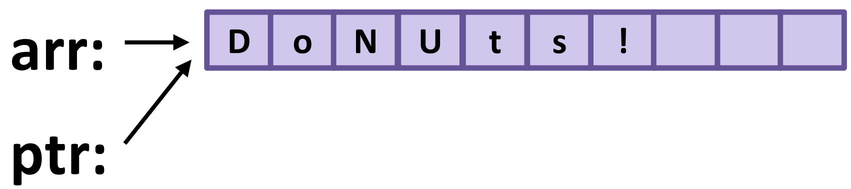
```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!
```

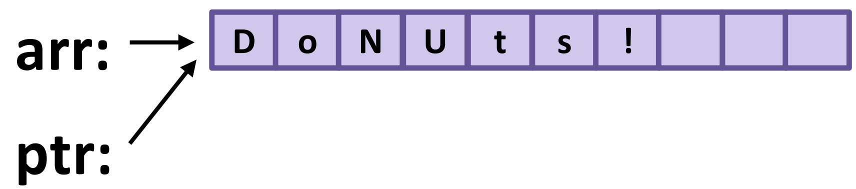
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
```

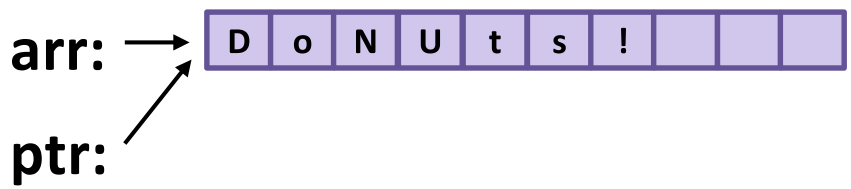
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
ptr DoNUts!
```

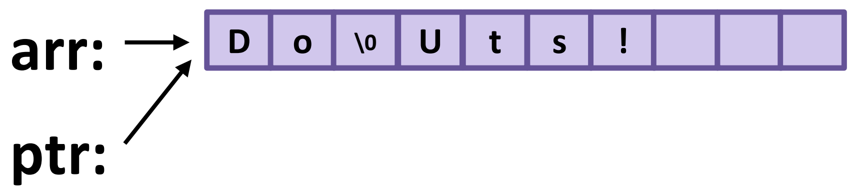
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
ptr DoNUts!
```

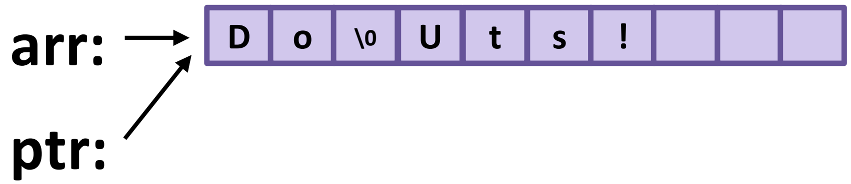
```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!

After uppercase
arr DoNUts!
ptr DoNUts!

After null terminator
arr Do
```

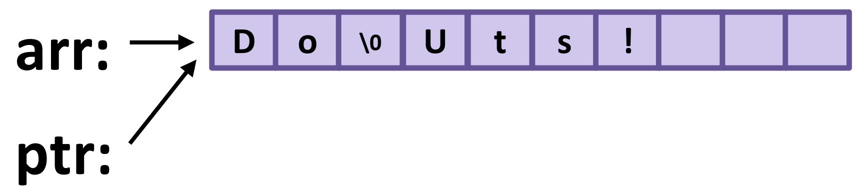
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!

After uppercase
arr DoNUts!
ptr DoNUts!

After null terminator
arr Do
ptr Do
```

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  printf("address of arr %p\n", &arr);
  printf("ptr string %s\n", ptr);
  printf("ptr pointer %p\n", ptr);
  printf("address of ptr %p\n", &ptr);

  ptr = arr;
  printf("\nAfter ptr = arr\n");
  printf("arr %s\n", arr);
  printf("address of arr %p\n", &arr);
  printf("ptr string %s\n", ptr);
  printf("*ptr char %c\n", *ptr); //same as arr[0]!
  printf("ptr pointer %p\n", ptr);
  printf("address of ptr %p\n", &ptr);


  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  *ptr = 'B';
  printf("\nAfter null terminator\n");
  printf("arr+2 %s\n", arr+3);
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```

```
[jeannie@rugger9-2021 slides % ./donuts
arr Donuts!
address of arr 0x16b25b7ce
ptr string (null)
ptr pointer 0x0
address of ptr 0x16b25b7b8

After ptr = arr
arr Donuts!
address of arr 0x16b25b7ce
ptr string Donuts!
*ptr char D
ptr pointer 0x16b25b7ce
address of ptr 0x16b25b7b8

After uppercase
arr DoNUts!
ptr DoNUts!

After null terminator
arr+2 Uts!
arr Bo
ptr Bo
```

Rule of thumb about pointers:

> **After declaring a pointer, only use the * again when trying to “dereference” or “follow” the pointer**