CS 61C
Great Ideas in Computer Architecture
Lecture 3: *Introduction to C, Part II*

Instructor: Sagar Karandikar

sagark@eecs.berkeley.edu

http://inst.eecs.berkeley.edu/~cs61c

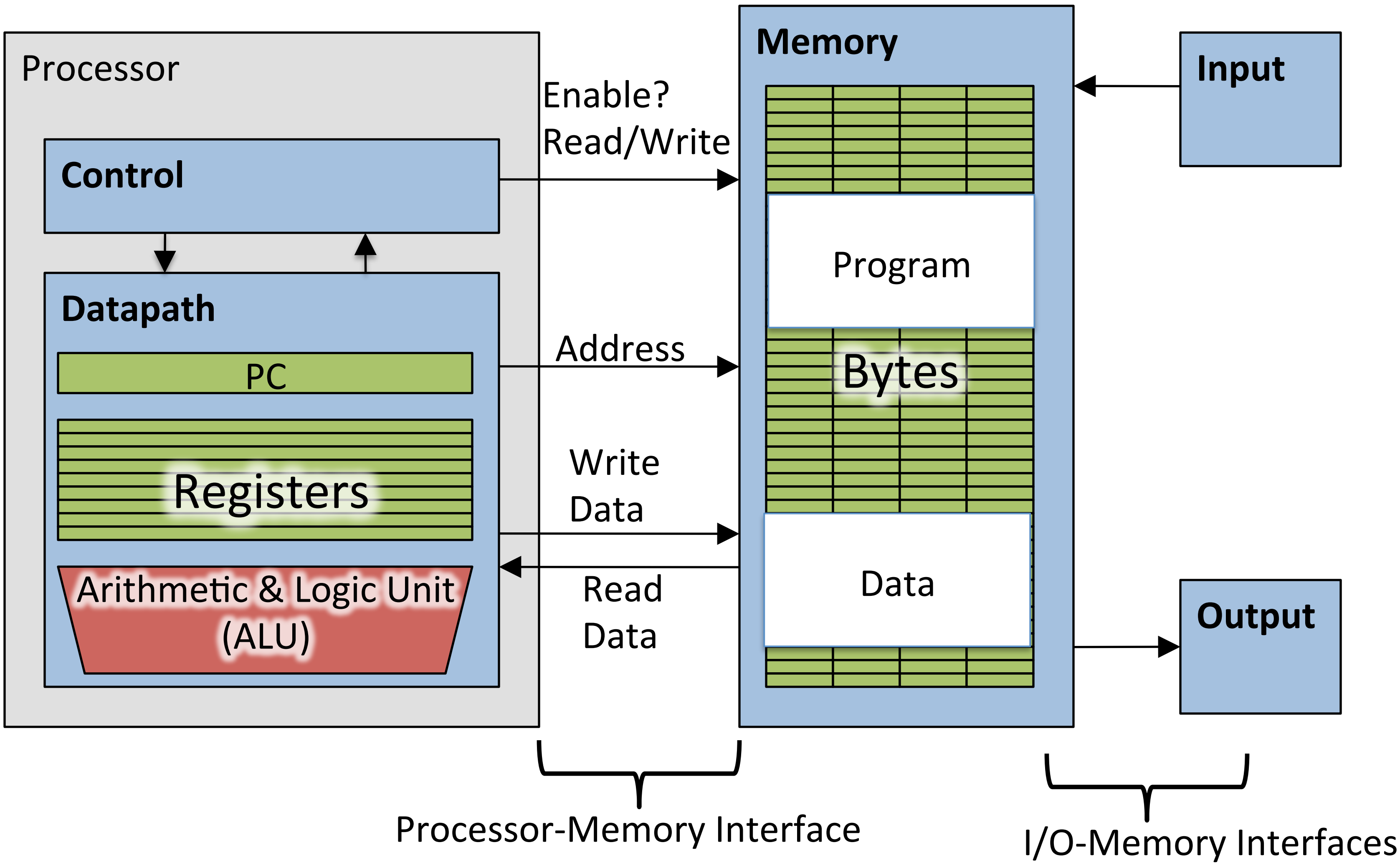
Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

New on the Schedule

- My Office Hours:
  - M 8-9am, W 11am-12pm, 511 Soda

    ^ today, right after lecture
  - Or by appointment
- Office hours:
  - My OH will be for conceptual questions – lectures, homework, exams
  - TA OH for project questions, lab checkoffs, etc.
  - We will rearrange TA OH to be later in the week for project/homework purposes

Review: Components of a Computer



http://en.cppreference.com/w/c/language/operator_precedence

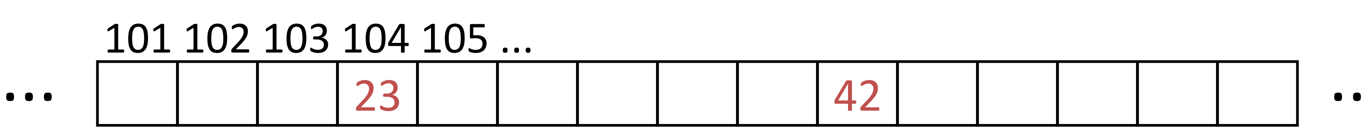
Review: C Operators - Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*) { *list* }	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	

...

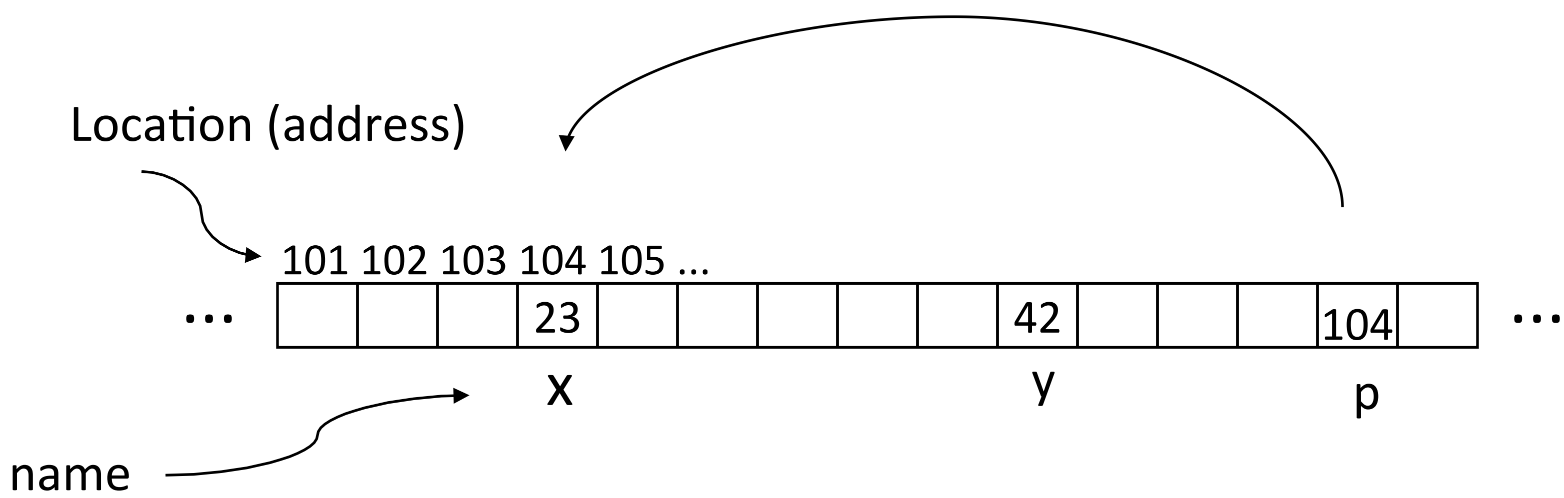
Review: Address vs. Value

- Consider memory to be a single huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don't confuse the address referring to a memory location with the value stored there



Review: Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable



Review: Pointer Syntax

- `int *x;`
  - Tells compiler that variable x is address of an `int`
- `x = &y;`
  - Tells compiler to assign address of `y` to `x`
  - `&` called the “address operator” in this context
- `z = *x;`
  - Tells compiler to assign value at address in `x` to `z`
  - `*` called the “dereference operator” in this context

Review: Creating and Using Pointers

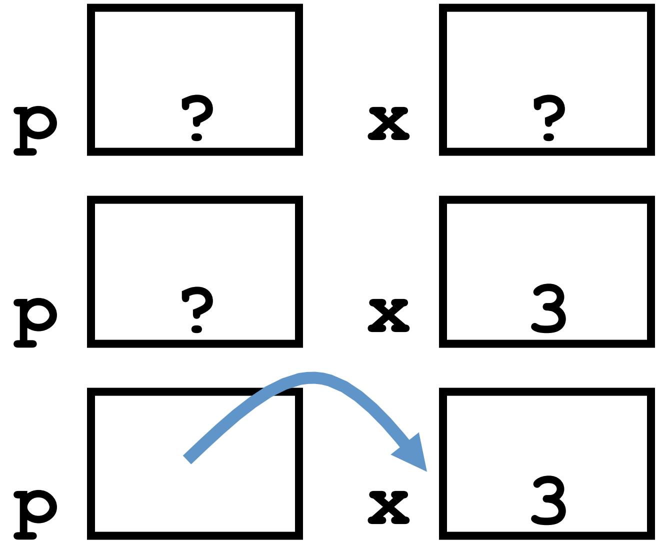
- How to create a pointer:

  & operator: get address of a variable

```
int *p, x;

    x = 3;

    p = &x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

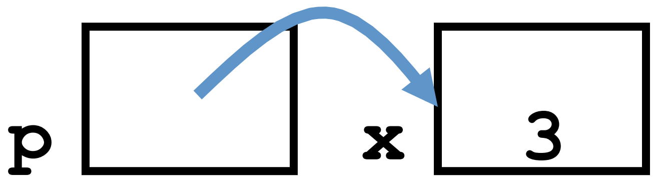
- How get a value pointed to?

  "*" (dereference operator): get the value that the pointer points to

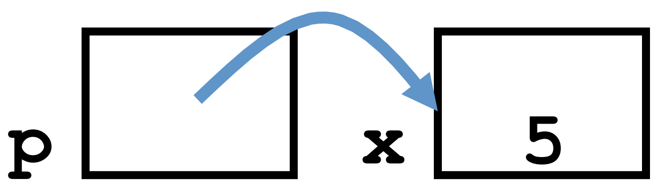
```
printf("p points to %d\n",*p);
```

Review: Using Pointer for Writes

- How to change a variable pointed to?
  - Use the dereference operator `*` on left of assignment operator `=`



`*p = 5;`



Review: Pointers and Parameter Passing

- Java and C pass parameters “by value”
  - Procedure/function/method gets a copy of the parameter, *so changing the copy cannot change the original*

```
void add_one (int y) {
    y = y + 1;
  }
int y = 3;
add_one(y);
```

y remains equal to 3

Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
  - Want to modify an object, not just pass its value
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—coming up next lecture
    - *Dangling references* and *memory leaks*

Video: Fun with Pointers

https://www.youtube.com/watch?v=6pmWojisM_E

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compiler hints which registers to use!
- Today, many applications attain acceptable performance using higher-level languages without pointers
- Low-level system code still needs low-level access via pointers, hence continued popularity of C

Clickers/Peer Instruction Time

```
void foo(int *x, int *y)
{   int t;
    if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);
```

Result is:

A: **a=3 b=2 c=1**

B: **a=1 b=2 c=3**

C: **a=1 b=3 c=2**

D: **a=3 b=3 c=3**

E: **a=1 b=1 c=1**

Administrivia

- HW0 out, everyone should have been added to edX
  - Due: Sunday @ 11:59:59pm
- HW0-mini-bio posted on course website
  - Give paper copy to your TA in lab next Tuesday
- First lab due at the beginning of your lab tomorrow
- Get Clickers!
- Let us know about exam conflicts by the end of this week

Break

C Arrays

- Declaration:

  ```
  int ar[2];
  ```

  declares a 2-element integer array: just a block of memory

  ```
  int ar[] = {795, 635};
  ```

  declares and initializes a 2-element integer array

C Chars

- char is a special type designed to hold all possible values of "execution character set"
- Standard mandates that sizeof(char) is 1 byte
- Byte is not guaranteed to be 8 bits, but we will assume so (and all modern archs do so)
- The character set modern systems use is the American Standard Code for Information Interchange (ASCII)

C Chars and ASCII

- Everything is still a number!
- ASCII maps characters to numerical values

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

C Chars and ASCII

- Everything is still a number!
- ASCII maps characters to numerical values
- Not great for non-western languages
- Modern apps use unicode
  - Considerably more complicated than ASCII
- Characters are symbols in single quotes:

```
char hello = ‘B’;
```

- Characters are distinct from strings!

C Strings

- String in C is just an array of characters

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (aka “null terminator”)

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

C String Standard Functions

- `int strlen(char *string);`
  - Computes the length of a NULL terminated string
- `int strcmp(char *str1, char *str2);`
  - Returns 0 if str1 and str2 are identical
  - What happens if you check the cond. str1 == str2?
- `char *strcpy(char *dst, char *src);`
  - Copy the contents of string src to the memory that dst points to
  - Caller's job to ensure that space dst points to is large enough
- More: http://www.cplusplus.com/reference/cstring/

Array Name / Pointer Duality

- *Key Concept*: Array variable is a "pointer" to the first (0^{th}) element
- So, array variables almost identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
- Consequences:
  - `ar` is an array variable, but looks like a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - Can use pointer arithmetic to conveniently access arrays

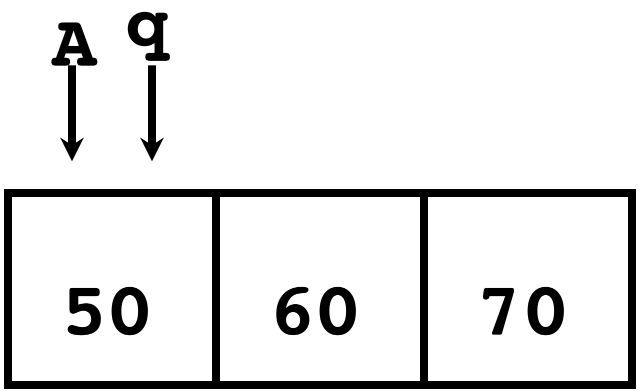
Changing a Pointer Argument?

- What if want function to change a pointer?
- What gets printed?

```
void inc_ptr(int *p)
{    p =  p + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
inc_ptr( q);
printf(“*q = %d\n”, *q);
```

*q = 50



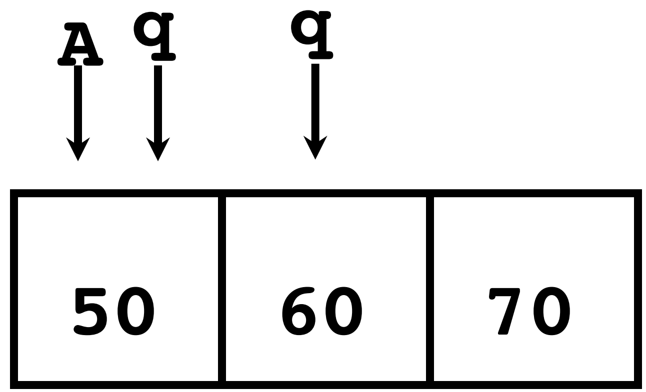
Pointer to a Pointer

- Solution! Pass a pointer to a pointer, declared as `**h`
- Now what gets printed?

```
void inc_ptr(int **h)
{    *h = *h + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```

`*q = 60`



C Arrays are Very Primitive

- An array in C does not know its own length, and its bounds are not checked!
  - Consequence: We can accidentally access off the end of an array
  - Consequence: We must pass the array *and its size* to any procedure that is going to manipulate it
- Segmentation faults and bus errors:
  - These are VERY difficult to find;
    be careful! (You'll learn how to debug these in lab)

Segmentation Faults vs. Bus Errors

```
1. 10:0:~/scratch – "[~/scratch] :zsh" (tmux)
------------------------------------------------------------
~/scratch » cat test.c                          sagar@dark-doodad
int main() {
    int *p;
    *p = 1000;
}
------------------------------------------------------------
~/scratch » gcc test.c                          sagar@dark-doodad
------------------------------------------------------------
~/scratch » ./a.out                             sagar@dark-doodad
[1]    76909 bus error  ./a.out
------------------------------------------------------------
~/scratch »                                     sagar@dark-doodad

[10] 0:~/scratch*                  "[~/scratch] :zsh" 21:51 23-Jun-15
```

Segmentation Faults vs. Bus Errors

```
1. 10:0:~/scratch – "[~/scratch] :zsh" (tmux)
-------------------------------------------------------------
~/scratch » cat test.c                        sagar@dark-doodad
int main() {
    int *p = 1003;
    *p = 1000;
}
-------------------------------------------------------------
~/scratch » gcc test.c                        sagar@dark-doodad
test.c:2:10: warning: incompatible integer to pointer conversion initializing
      'int *' with an expression of type 'int' [-Wint-conversion]
    int *p = 1003;
         ^   ~~~~
1 warning generated.
-------------------------------------------------------------
~/scratch » ./a.out                           sagar@dark-doodad
[1]    77613 segmentation fault  ./a.out
-------------------------------------------------------------
~/scratch »                                   sagar@dark-doodad

[10] 0:~/scratch*                "[~/scratch] :zsh" 21:55 23-Jun-15
```

Segmentation Faults vs. Bus Errors

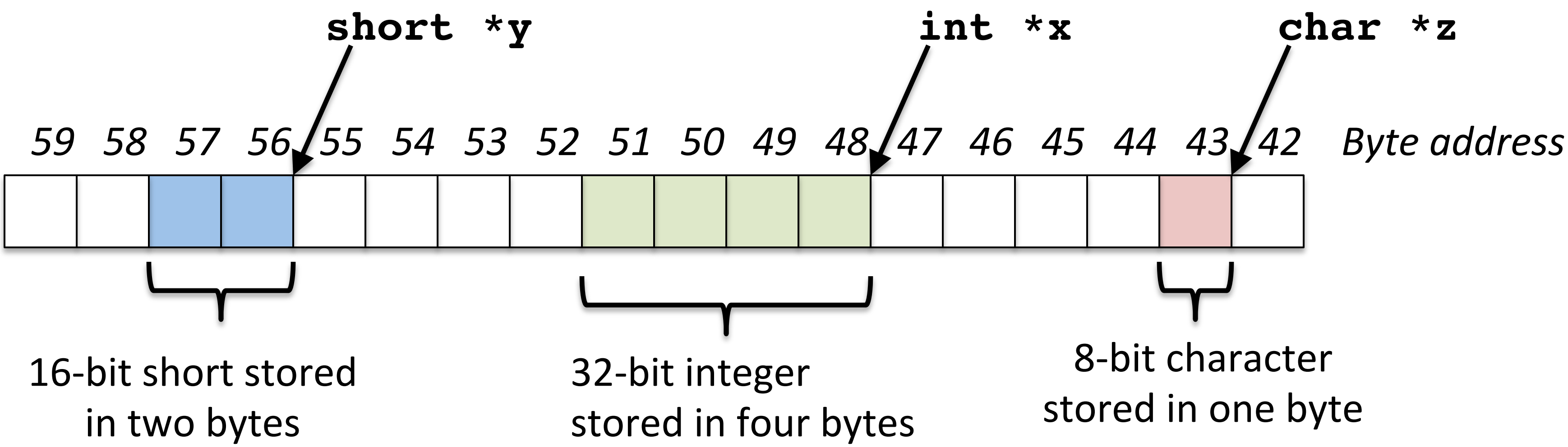
- Both are common errors when using pointers, but segfaults are most common
- **Bus Error**: Your program is trying to do something that the hardware bus does not support
  - e.g. unaligned accesses
- **Segmentation Fault**: Your program is trying to access memory that it doesn't have permission to modify
  - e.g. accidentally walking off the end of an array

Use Defined Constants

- Array size *n*; want to access from *0* to *n-1*, so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern
    ```
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Better pattern
    ```
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```
- Accessing elements:
  ```
  ar[num]
  ```
- SINGLE SOURCE OF TRUTH
  - You're utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: "Don't Repeat Yourself"

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  - Hardware's memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes



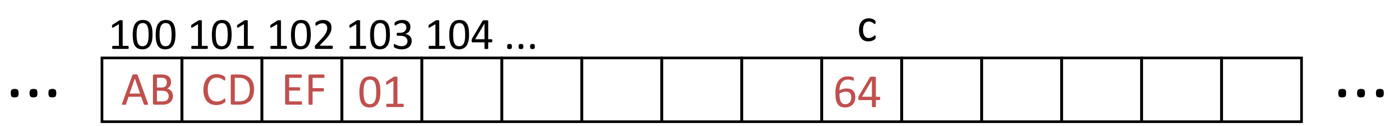
Endianness

- Not clear which order to store bytes that make up an int (or any > 1 byte quantity) in memory
- Big-endian:
  - store the most significant byte of a word at the smallest memory address
- Little-endian:
  - store the least significant byte of a word at the smallest memory address

Big-Endian

- Say we want to store 0xABCDEF01 in memory:

```
int a = 0xABCDEF01
int b[1];
int *c = b;
c[0] = a;
```

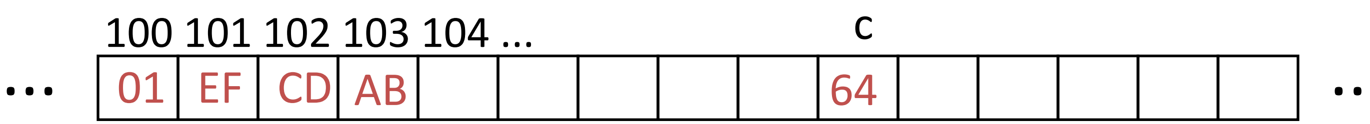


Addresses in decimal, values in hex

Little-Endian

- Say we want to store 0xABCDEF01 in memory:

```
int a = 0xABCDEF01
int b[1];
int *c = b;
c[0] = a;
```



Addresses in decimal, values in hex

sizeof() operator

- sizeof(type) returns number of bytes in object
  - But number of bits in a byte is not standardized
    - In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
- By definition, sizeof(char)==1
- Can take sizeof(arr), or sizeof(structtype)
- We'll see more of sizeof when we look at dynamic memory management

Pointer Arithmetic

pointer + *number* *pointer* – *number*

e.g., *pointer* **`+ 1`** adds 1 <u>something</u> to a pointer

```
char    *p;
char     a;
char     b;

p = &a;
p += 1;
```

In each, p now points to b
(Assuming compiler doesn't reorder variables in memory.
Never code like this!!!!)

```
int    *p;
int     a;
int     b;

p = &a;
p += 1;
```

Adds **`1*sizeof(char)`** to the memory address

Adds **`1*sizeof(int)`** to the memory address

Pointer arithmetic should be used <u>cautiously</u>

Arrays and Pointers

- Array ≈ pointer to the initial (0th) array element

$$a[i] \equiv *(a+i)$$

- An array is passed to a function as a pointer
  - The array size is lost!

- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

Passing arrays:

Really `int *array`

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

... because **`array`** is really a pointer (and a pointer is architecture dependent, but likely to be 8 on modern machines!)

What does this print? **40**

Arrays and Pointers

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

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These code sequences have the same effect!

Clickers/Peer Instruction Time

```
int x[5] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);
```

Result is:

A: 2

B: 3

C: 4

D: 5

E: None of the above

CS61C In the News (1/23/2015): Google Exposing Apple Security Bugs

- Google security published details of three bugs in Apple OS X (90 days after privately notifying Apple)
  - One network stack problem fixed in Yosemite
  - One is dereferencing a null pointer !
  - One is zeroing wrong part of memory !
- Separately, Google announces it won't patch WebKit vulnerability affecting Android 4.3 and below (only about 930 million active users)

Break

Concise strlen()

```
int strlen(char *s)
{
    char *p = s;
    while (*p++)
        ; /* Null body of while */
    return (p - s - 1);
}
```

What happens if there is no zero character at end of string?

Point past end of array?

- Array size n; want to access from 0 to $n-1$, but test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
 ...
 p = &ar[0]; q = &ar[10];
 while (p != q)
    /* sum = sum + *p; p = p + 1; */
  sum += *p++;
```

  – Is this legal?
- C defines that one element past end of array must be a valid address, i.e., not cause an error

Valid Pointer Arithmetic

- Add an integer to a pointer.
- Subtract 2 pointers (in the same array)
- Compare pointers (<, <=, ==, !=, >, >=)
- Compare pointer to NULL (indicates that the pointer points to nothing)

Everything else illegal since makes no sense:

- adding two pointers
- multiplying pointers
- subtract pointer from integer

Arguments in `main()`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[])`
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:

    unix% sort myFile
  - `argv` is a *pointer* to an array containing the arguments as strings

Example

- `foo hello 87`
- `argc = 3 /* number arguments */`
- `argv[0] = "foo",`
  `argv[1] = "hello",`
  `argv[2] = "87"`
  - Array of pointers to strings

Clickers/Peer Instruction Time

How many of the following pointer ops will generate warnings or errors in the compiler?

1. pointer + integer
2. integer + pointer
3. pointer + pointer
4. pointer - integer
5. integer - pointer
6. pointer - pointer
7. compare pointer to pointer
8. compare pointer to integer
9. compare pointer to 0
10. compare pointer to NULL

<u># invalid</u>

a) 1
b) 2
c) 3
d) 4
e) 5

And In Conclusion, …

- Pointers are abstraction of machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type of the object they point to (except void *)
- Pointers are powerful but potentially dangerous