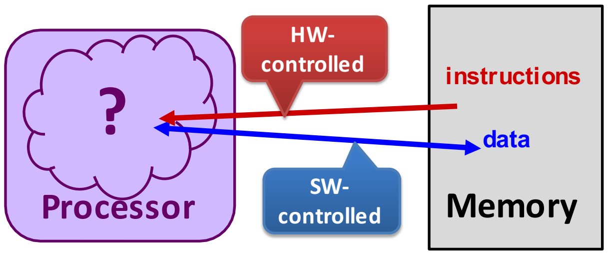
Modern Digital Computer (from the outside)

"von Neumann" model



How does a program find its data in memory?

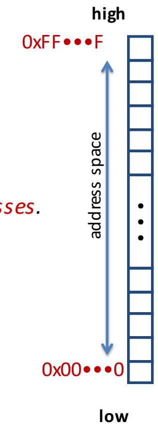
Byte-Addressable Memory

Memory = array of byte *locations*,
unique *address* = index.

Read/Write

Programs refer to bytes in memory by their *addresses*.

Address = word
Address space = range of possible addresses



Words in Memory

Address of word
= address of 1st byte in word

Alignment

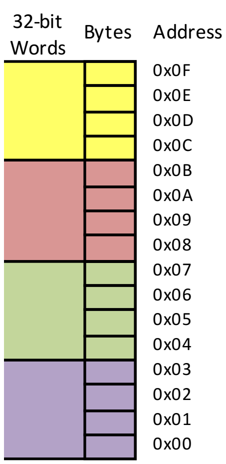
Data of size *n bytes* stored at *a*
only if *a* mod *n* = 0

n is a power of 2

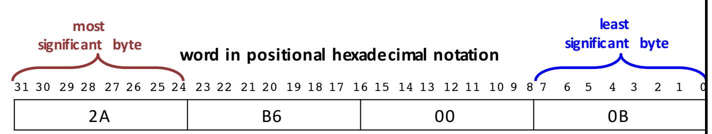
Recommended (x86), or required (MIPS) depending on platform.

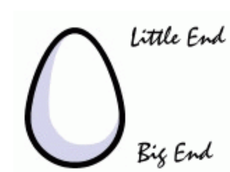
Why?

Byte ordering: what is the *"1st"* byte in a word?



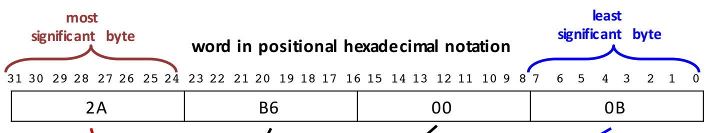
Endianness:
byte order *within memory words*





Bit order within bytes is always the same.

little endian *(we **will** use this)*



Little-endian memory layout.

Address	Contents
03	2A
02	B6
01	00
00	0B

used by **x86**

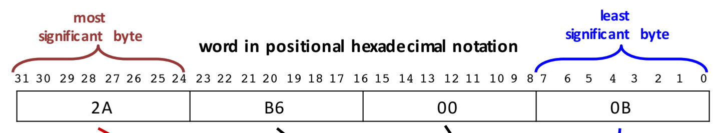
Little end first:

Most significant byte at highest address.

Position increases as address increases.

Least significant byte at lowest address.

big endian *(we will **not** use this)*



Big end first:

Least significant byte at highest address.

Position decreases as address increases.

Most significant byte at lowest address.

Big-endian memory layout.

Address	Contents
03	0B
02	00
01	B6
00	2A

used by networks, PowerPC, MIPS, Sparc

Little Endianness in Machine Code

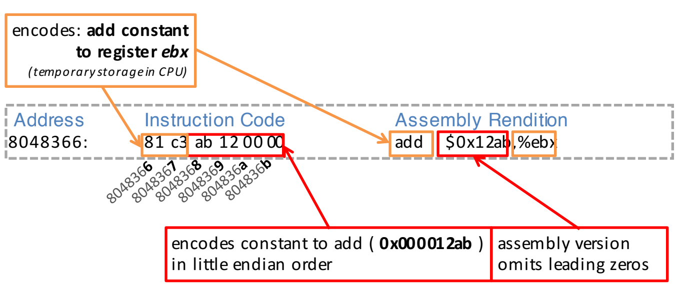
Disassembly

Take binary machine code and generate an assembly code version.

Instruction as stored in memory

Shows byte encoding of instruction as stored in memory,

with byte in lower address on left and byte in higher address on right.



When does endianness matter?

Mostly invisible most of the time.

Matters only when inspecting memory byte-by-byte.

For now: **endianness matters ONLY IN MEMORY.**

Memory stores bytes, so must define how to split larger values into bytes.

It also matters on the network or in files.

Byte order within word is always natural within the processor.

Processor manipulates entire words, so need to split them up.

Bit order within bytes is always natural.

Addresses and Pointers

address **= number of location in memory**

pointer **= data object that holds an address**

The value 240 is stored at address 0x20.

$240_{10} = F0_{16} = $ 0x00 00 00 F0

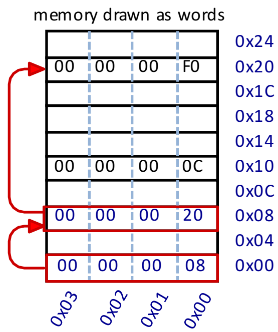
A **pointer** stored at address 0x08 points to address 0x20.

A **pointer to a pointer** is stored at address 0x00.

The **value** 12 is stored at address 0x10.

Is it a pointer?

Are any of these values pointers?



Data Representations

Sizes of data types (in bytes)

Java Data Type	C Data Type	32-bit word	64-bit word
boolean	*bool*	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	(pointer) *	4	8

address size = word size

Addresses and Pointers in C

& = 'address of'

** = 'contents at address' or 'dereference'*

```
int* ptr;
```

Declare a variable, `ptr`, that is a pointer to (*i.e.*, holds the address of) an int in memory.

```
int x = 5;
int y = 2;
```

Declare two variables, `x` and `y`, that hold ints, and sets them to 5 and 2, respectively.

```
ptr = &x;
```

Set `ptr` to the address of `x`. Now, "*`ptr` points to `x`*"

"Dereference `ptr`."

```
y = 1 + *ptr;
```

Set `y` to: 1 plus the value at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y=1+x;`

What is `*(&y)` ?

Writing pointer types

Spaces between base type, *, and variable name do not matter. The following are equivalent:

```
int* ptr;
```

I prefer this

Suggests: "The variable **ptr** holds the **address of an int** in memory."

```
int *ptr;
```

will see this a lot in others' code

Suggests: "There is an **int** in memory at the **address held by** the variable **ptr**."

Caveat: do not declare multiple variables on same line if using the former.

Assignment in C

& = 'address of'
* = 'contents at address' or 'dereference'

A variable is represented by a memory location.

Initially, it may hold any value.

int x, y;

// x is at location 0x20, y is at 0x0C.

A7	00	32	00	0x24	
00	01	29	F3	0x20	x
EE	EE	EE	EE	0x1C	
FA	CE	CA	FE	0x18	
26	00	00	00	0x14	
00	00	10	00	0x10	
01	00	00	00	0x0C	y
FF	00	F4	96	0x08	
00	00	00	00	0x04	
00	42	17	34	0x00	

14

Assignment in C

& = 'address of'
* = 'contents at address' or 'dereference'

Left-hand-side = right-hand-side;

LHS must evaluate to a *place to store a value.*

RHS must evaluate to a *value.*

Store RHS value at LHS location.

int x, y;

x = 0;

y = 0x3CD02700;

x = y + 3;

// Get value at y, add 3, put it in x.

int* z = &y;

// Get address of y, put it in z.

***z = y;**

// What does this do?

				0x24	
				0x20	x
				0x1C	
				0x18	
				0x14	
				0x10	
				0x0C	y
				0x08	
				0x04	z
				0x00	

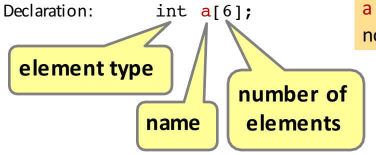
15

Arrays in C

Arrays are adjacent locations in memory storing the same type of data object.

a is a name for the array's address, not a pointer to the array.

Declaration: `int a[6];`



				0x24
				0x20
				0x1C
				0x18
				0x14
				0x10
				0x0C
				0x08
				0x04
				0x00

Arrays in C

Arrays are adjacent locations in memory storing the same type of data object.

a is a name for the array's address, not a pointer to the array.

The address of a[i] is the address of a[0] plus i times the element size in bytes.

Declaration: `int a[6];`

Indexing:
```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:
```
int* p;
p = a;
p = &a[0];
*p = 0xA;
```
(equivalent: `p = a;` and `p = &a[0];`)

equivalent:
```
p[1] = 0xB;
*(p + 1) = 0xB;
p = p + 2;
```

array indexing = address arithmetic
Both are scaled by the size of the type.

```
*p = a[1] + 1;
```

				0x24	
				0x20	a[5]
				0x1C	
				0x18	
				0x14	…
				0x10	
				0x0C	a[0]
				0x08	
				0x04	p
				0x00	

Array Allocation

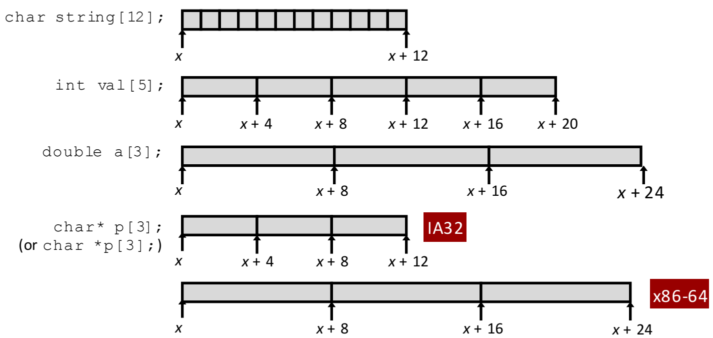
Basic Principle

`T A[N];`

Array of data type `T` and length `N`

Contiguously allocated region of *N* * `sizeof(T)` bytes

Use this to determine proper size in C.



27

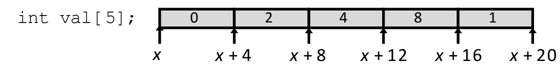
Array Access

Basic Principle

`T A[N];`

Array of data type `T` and length `N`

Identifier `A` can be used as a pointer to array element 0: Type *T**



Reference	Type	Value
val[4]	int	
val	int *	
val+1	int *	
&val[2]	int *	
val[5]	int	
*(val+1)	int	
val + i	int *	

28

Representing strings

A C-style string is represented by an array of bytes (*char*).

— Elements are one-byte ASCII codes for each character.

— ASCII = American Standard Code for Information Interchange

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated Strings

- C strings are arrays of characters ending with the *null* character.

Why?

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- Compute the string length.
- Does Endianness matter for strings?

* vs []

Since

- array name == address of 0th element
- array indexing == pointer arithmetic

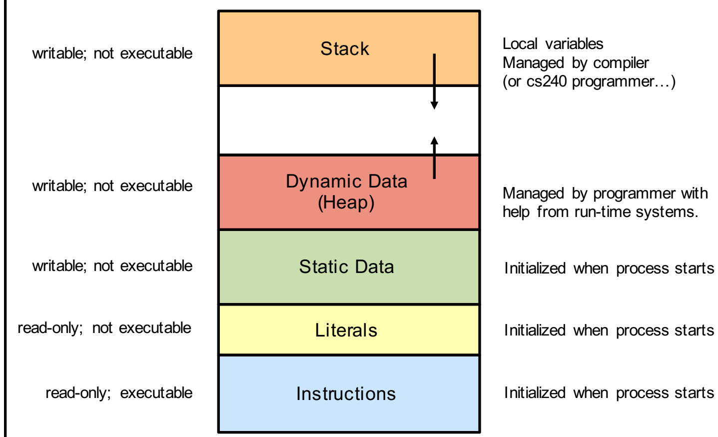
C programmers often use * where you might expect []:

- e.g.: char* is a:
  - pointer to a char
  - pointer to the first char in a string of unknown length

int main(int argc, char argv);**

int strcmp(char* a, char* b);

Memory Layout



38

Dynamic memory allocation

`#include <stdlib.h>`

`void* malloc(size_t size)`

Successful:

Returns a pointer to a memory block of at least `size` bytes (typically) aligned to 8-byte boundary

If `size == 0`, returns NULL

Unsuccessful: returns NULL and sets `errno`

`void free(void* p)`

Returns the block pointed at by `p` to pool of available memory

`p` must come from a previous call to `malloc`

39

Malloc/free Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
  p = (int *)malloc(n * sizeof(int));
  if (p == NULL) {
    perror("malloc");  // print an error message
    exit(0);
  }
  for (i=0; i<n; i++) p[i] = i;

  free(p); /* return p to available memory pool */
}
```

malloc rules:

cast result to proper pointer type

Use **sizeof(...)** to determine size

free rules:

Free only objects acquired from malloc, and only once.

Do not use an object after freeing it.

40

http://xkcd.com/138/

Memory-Related Perils and Pitfalls in C

!!!

(Terrible things to do with pointers, part 1.)

Dereferencing bad pointers

See lab exercises for:

- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks

42

Scanf: read formatted input

```
int val;
...
scanf("%d", &val);
```

Declared, but not initialized – holds anything.

Read one int from input.

Store it in memory at this address

i.e., store it in memory at the address **where the contents of val is stored**: store into memory at 0xFFFFFF38.

		0xFFFFFF3C
val	BA D4 FA CE	0xFFFFFF38
		0xFFFFFF34

43

The classic scanf bug

!!!

Forget one symbol... unleash certain doom.

```
int val;
...
scanf("%d", val);
```

Declared, but not initialized – holds anything.

Read one int from input.

Store it in memory at this address

i.e., store it in memory at the address **given by the contents of val**: store into memory at 0xBAD4FACE.

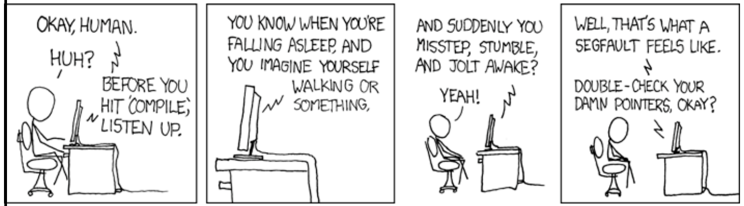
		0xFFFFFF3C
val	BA D4 FA CE	0xFFFFFF38
		0xFFFFFF34
	...	...
	CA FE 12 34	0xBAD4FACE

Best case: segmentation fault, or bus error, crash.

Worst case: silently corrupt data stored at address 0xBAD4FACE, and val still holds 0xBAD4FACE.

44

C memory error messages



http://xkcd.com/371/

11: segmentation fault

accessing address outside legal area of memory

10: bus error

accessing misaligned or other problematic address

Practice debugging in lab!

Why C?

Why learn C?

- Think like actual computer: abstraction very close to machine level.
- Understand just how much Your Favorite Language provides.
- Understand just how much Your Favorite Language might cost.
- Classic.
- Still (more) widely used (than it should be).
- Pitfalls still fuel many security vulnerabilities, devastating bugs today.

Why not use C?

- Almost definitely not the right language for your next personal project.
- It "gets out of the programmer's" way even when the programmer is running towards a blind cliff.
- Many advances in other programming languages since then fix a lot of its problems while keeping strengths.