Strings and Memory Representations

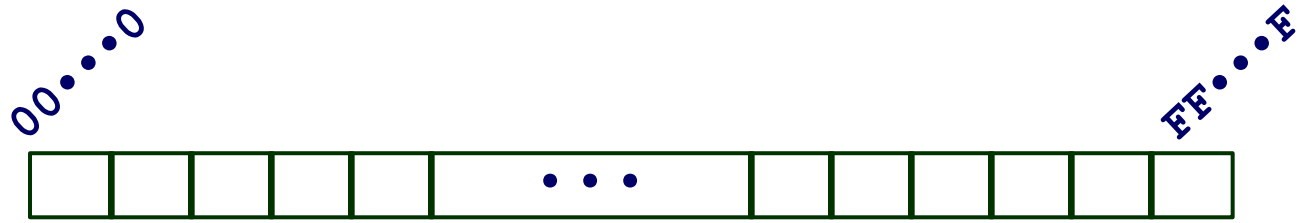
Reading: CS:APP Chapter 2.1

These slides adapted from materials provided by the textbook authors.

Strings and Memory Representations

- From Bytes to Words
- Byte ordering
- Pointer representation
- Character and string representation
- Summary

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

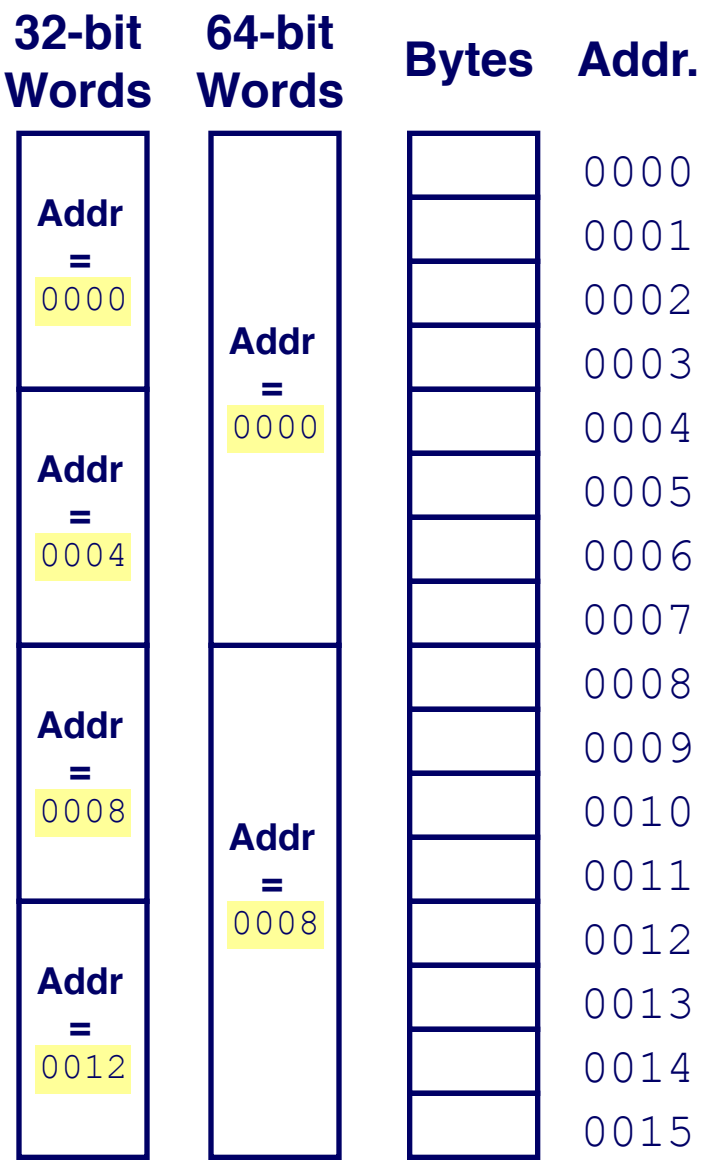
- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, 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
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



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

Byte Ordering

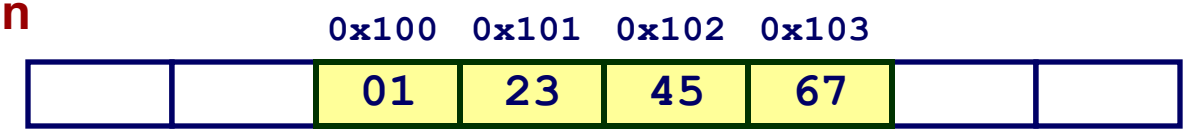
- **So, how are the bytes within a multi-byte word ordered in memory?**

- **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

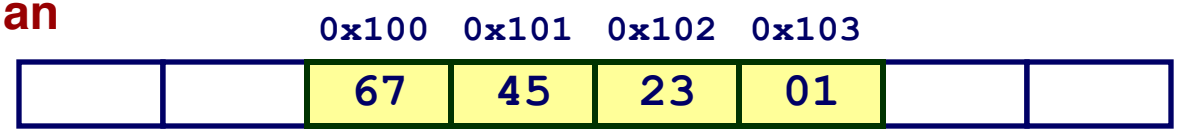
Byte Ordering Example

- **Example**
  - Variable Y has 4-byte value of 0x01234567
  - Address given by &Y is 0x100

Big Endian



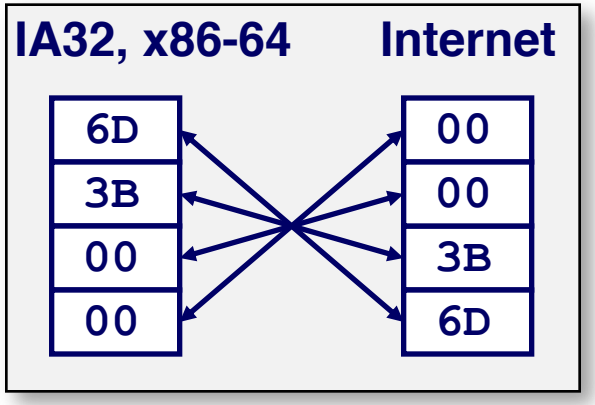
Little Endian



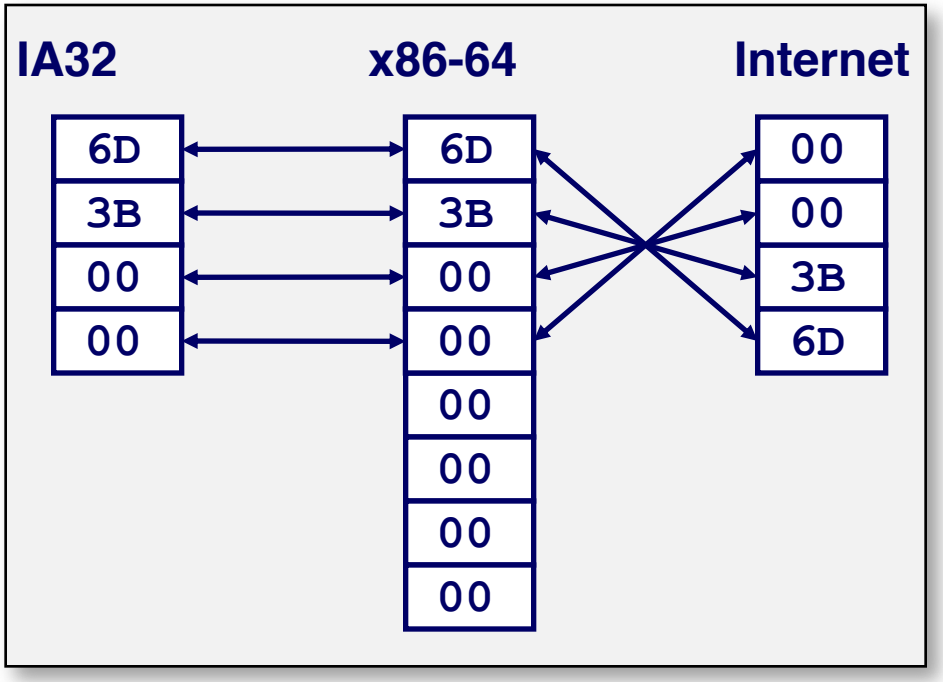
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

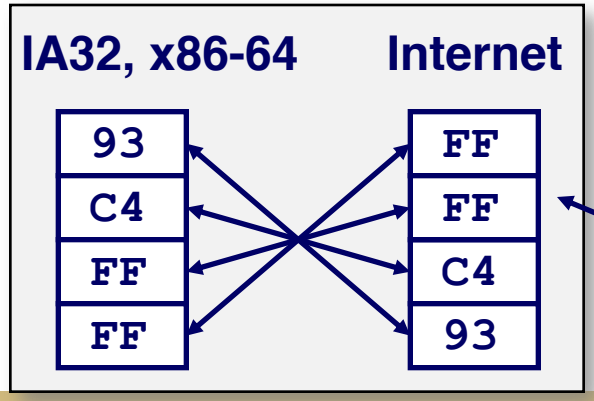
```
int A = 15213;
```




```
long int C = 15213;
```




```
int B = -15213;
```



Two's complement representation

Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf(”%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

show_bytes Execution Example

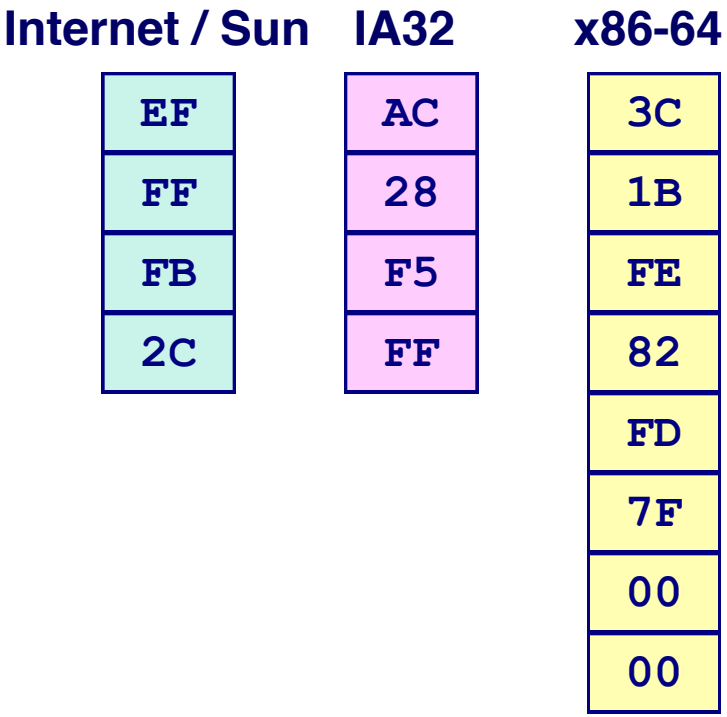
```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

```
int B = -15213;
int *P = &B;
```



Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Characters

- Representing letters and words by numbers: morse code
- Later TTY systems converted typewriter into morse / baudot other codes
- Led to ASCII (American Standard Code for Information Interchange) in ~1960
- 7 (then 8) bit code for letters, numbers, "actions"
- Expanded to Unicode in 90's and UTF-8 in '00's

ف ஔ ఊ ఋ

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ASCII Table == First 127 of UTF-8

- Alphabetic
- Control character
- Numeric digit
- Punctuation
- Extended punctuation
- Graphic character
- International
- Undefined

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000 *0*	SOH 0001 *1*	STX 0002 *2*	ETX 0003 *3*	EOT 0004 *4*	ENQ 0005 *5*	ACK 0006 *6*	BEL 0007 *7*	BS 0008 *8*	HT 0009 *9*	LF 000A *10*	VT 000B *11*	FF 000C *12*	CR 000D *13*	SO 000E *14*	SI 000F *15*
1_	DLE 0010 *16*	DC1 0011 *17*	DC2 0012 *18*	DC3 0013 *19*	DC4 0014 *20*	NAK 0015 *21*	SYN 0016 *22*	ETB 0017 *23*	CAN 0018 *24*	EM 0019 *25*	SUB 001A *26*	ESC 001B *27*	FS 001C *28*	GS 001D *29*	RS 001E *30*	US 001F *31*
2_	SP 0020 *32*	! 0021 *33*	" 0022 *34*	# 0023 *35*	$ 0024 *36*	% 0025 *37*	& 0026 *38*	' 0027 *39*	(0028 *40*	) 0029 *41*	* 002A *42*	+ 002B *43*	, 002C *44*	- 002D *45*	. 002E *46*	/ 002F *47*
3_	0 0030 *48*	1 0031 *49*	2 0032 *50*	3 0033 *51*	4 0034 *52*	5 0035 *53*	6 0036 *54*	7 0037 *55*	8 0038 *56*	9 0039 *57*	: 003A *58*	; 003B *59*	< 003C *60*	= 003D *61*	> 003E *62*	? 003F *63*
4_	@ 0040 *64*	A 0041 *65*	B 0042 *66*	C 0043 *67*	D 0044 *68*	E 0045 *69*	F 0046 *70*	G 0047 *71*	H 0048 *72*	I 0049 *73*	J 004A *74*	K 004B *75*	L 004C *76*	M 004D *77*	N 004E *78*	O 004F *79*
5_	P 0050 *80*	Q 0051 *81*	R 0052 *82*	S 0053 *83*	T 0054 *84*	U 0055 *85*	V 0056 *86*	W 0057 *87*	X 0058 *88*	Y 0059 *89*	Z 005A *90*	[005B *91*	\ 005C *92*	] 005D *93*	^ 005E *94*	_ 005F *95*
6_	` 0060 *96*	a 0061 *97*	b 0062 *98*	c 0063 *99*	d 0064 *100*	e 0065 *101*	f 0066 *102*	g 0067 *103*	h 0068 *104*	i 0069 *105*	j 006A *106*	k 006B *107*	l 006C *108*	m 006D *109*	n 006E *110*	o 006F *111*
7_	p 0070 *112*	q 0071 *113*	r 0072 *114*	s 0073 *115*	t 0074 *116*	u 0075 *117*	v 0076 *118*	w 0077 *119*	x 0078 *120*	y 0079 *121*	z 007A *122*	{ 007B *123*	| 007C *124*	} 007D *125*	~ 007E *126*	DEL 007F *127*

ASCII collating order

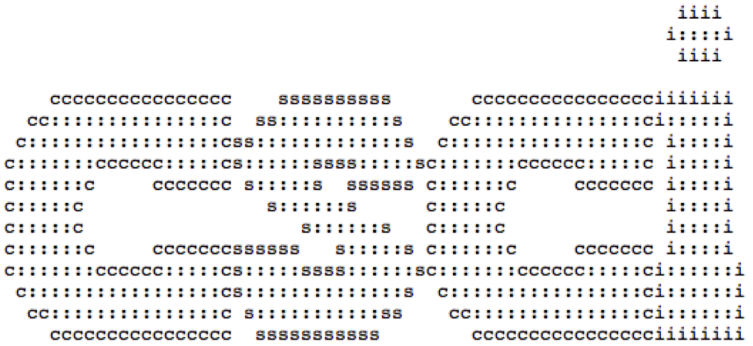
- Sequences 0..9, a..z and A..Z are in order
- Checking if ‘d’ is a digit:

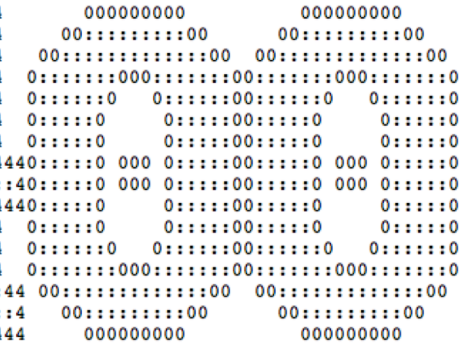
  (d >= ‘0’) && (d <= ‘9’)
- Checking if ‘l’ is a letter:

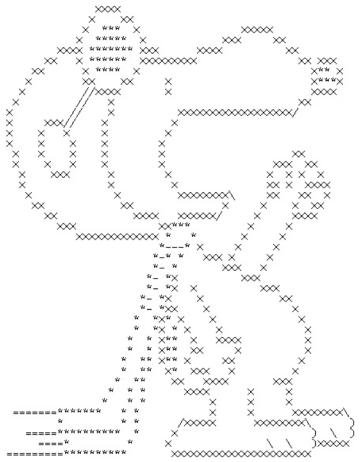
  ((l >= ‘a’) && (l <= ‘z’)) || ((l >= ‘A’) && (l <= ‘Z’))
- Change letter ‘A’..’Z’ to lower case:

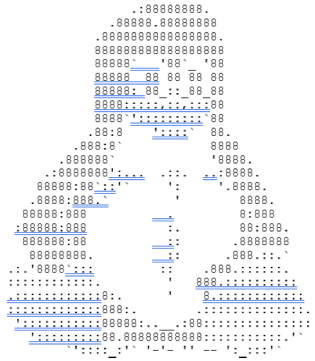
  (l – ‘A’) + ‘a’

Humans make art no matter how primitive the tools







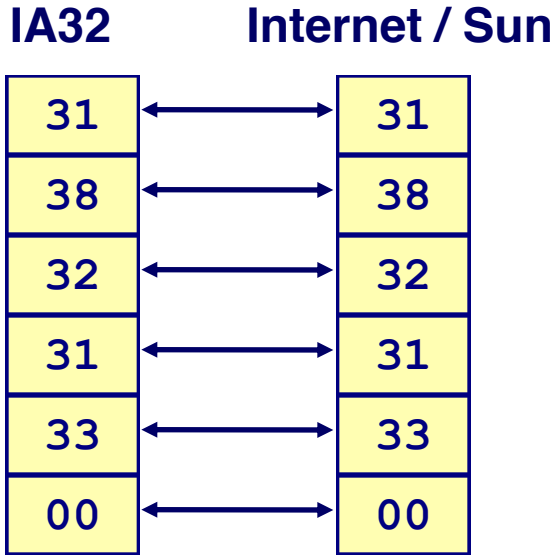


http://patorjk.com/software/taag/#p=display&f=Doh&t=csci%202400

Representing Strings

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

- **Strings in C**
  - Represented by array of ‘char’
  - Char ‘x’ enclosed in single quotes, string “enclosed in double”
  - ‘abcd’ is not same as “abcd”
    - ‘abcd’ is a 4–byte integer, not a string
    - “abcd” is 5 byte string with null terminator
  - Each character encoded in ASCII format
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue for strings



C sizeof() vs. string length vs. constant

- Common mistake is to think sizeof() returns the string length or the length of the ”buffer” that is pointed to

- A “char *” points to a char; convention is it’s the first char of a string but C doesn’t keep track of length

```
int main()
{
  char buffer[100];
  char *ptr;
  char *str = "how”;
  int   istr = 'how’;
  printf("sizeof(buffer) = %d\n", sizeof(buffer));
  printf("sizeof(ptr)    = %d\n", sizeof(ptr));
  printf("sizeof(*ptr)   = %d\n", sizeof(*ptr));
  printf("sizeof(str)    = %d\n", sizeof(str));
  printf("sizeof(istr)   = %d\n", sizeof(istr));
}
```

String and Memory Functions in C

http://www.cplusplus.com/reference/cstring/

String

- strcpy(dst, src) – copy
- strlen(str) – length
- strcat(dest, src) – append
- strcmp(src1, src2)–compare

- strnlen(str, max)
- strncpy(dst, src, dstsize)
- strncat(dest, src, dstsize)
- strncmp(src1, src2, max)

Memory

- memcpy(dst, src, len)
- memmove(dst, src, len)
- memset(dst, value, len)
- memcmp(dst, src, len)

C string functions – strcpy 3 ways

- And all of them bad!

```
char * strcpy(char *dst, char *src){
   int i;
   for( i = 0; src[i] != '\0'; i++) {
     dst[i] = src[i];
   }
   dst[i] = 0;
   return dst;
}
```

```
char * strcpy(char *dst, char *src){
   char *orig = dst;
   for( ; *src ; dst++, src++ ) {
     *dst = *src;
   }
   *dst = 0;
   return orig;
}
```

```
char * strcpy(char *dst, char *s
   char *orig = dst;
   while(*src) {
     *dst++ = *src++;
   }
   *dst = 0;
   return orig;
}
```

C string functions – strncpy

- Most compilers will warn about strcpy, strlen, etc

```
char * strcpy(char *dst, char *src, size_t len){
   int i;

   for( i = 0; i < len && src[i] != '\0'; i++) {
     dst[i] = src[i];
   }
   dst[i] = 0;
   return dst;
}
```

- You'll be directed to the 'n' versions

C memory functions – memcpy

- The mem* functions don't look for null terminator

```
void* memcpy(void *dst, void *src, size_t len){
   int i;
   char *cdst = (char *) dst;
   char *csrc = (char *) src;
   for( i = 0; i < len ; i++) {
     cdst[i] = csrc[i];
   }
   return dst;
}
```

- Use provided str* and mem* functions rather than writing your own.
  - Much, much faster code
  - They already made it correct

C strcmp and memcmp

- strncmp(a,b) and memcmp(a,b,len) return -1, 0 or 1
  - -1 a < b
  - 0 0 == b
  - 1 a > b
- Leads to weird looking programming idiom

```
if ( ! strncmp(a,b,n)) {
   /* a is equal to b */
} else {
   /* a != b */
}
```

- Better to write this like..

```
if ( ! strncmp(a,b,n)== 0 ) {
   /* a is equal to b */
}
```

C memcpy and memmove

- **memcpy(a,b,len) and memmove(a,b,len) are similar**
  - memcpy will blindly copy ‘b’ to ‘a’ up to ‘len’ bytes
  - memmove acts as if data was copied to intermediate location. This is important if ‘a’ and ‘b’ overlap
- **Example: copying a buffer to make room in first element**

```
int buffer[N];
….
memmove(&buffer[1], &buffer[0], (N-1) * sizeof(int))
```

C strncat

- **strncat(a,b,len) appends ‘b’ to ‘a’**
  - Must have enough room in ‘a’

- **You can accomplish same goal using snprintf()**

```
char buffer[1024];
snprintf(a, len, “%s%s”, a, b)
```