Data in Memory

variables have multiple attributes

- symbolic name
- data type (perhaps with qualifier)
  - allocated in data area, stack, or heap
  - duration (lifetime or extent)
  - scope (visibility of the variable name)
  - linkage (may have been allocated elsewhere)
- storage class
- address in memory (none if only allocated in register)
- alignment
- byte order for multibyte variables
- initialization (optional)
- current value
simple data types are: char, int, float, double
(C uses the keyword void to denote untyped)

qualifiers: signed, unsigned, long, short, volatile
(compilers sometimes ignore "long", "short")
("volatile" is used to suppress optimizations)

data sizes
byte    -   8 bits = 1 byte  -  character
halfword - 16 bits = 2 bytes -  short integer
word    -  32 bits = 4 bytes -  integer, long integer in
          32-bit model
doubleword - 64 bits = 8 bytes - double, long integer in
             64-bit model
Data in Memory

```c
int main(){
    printf("sizeof(char) = %d\n", sizeof(char));
    printf("sizeof(short) = %d\n", sizeof(short));
    printf("sizeof(int) = %d\n", sizeof(int));
    printf("sizeof(long) = %d\n", sizeof(long));
    return 0;
}
```

On a CS machine:
compile with just gcc  compile with "gcc -m64"

    sizeof(char) = 1    sizeof(char) = 1
    sizeof(short) = 2   sizeof(short) = 2
    sizeof(int) = 4     sizeof(int) = 4
    sizeof(long) = 8    sizeof(long) = 8
Alignment in byte-addressable memory

- memory access is typically on word-by-word basis (aligned)
  (original meaning of "memory word" was the unit of transfer to/from memory)

- alignment restrictions on load/store instructions prevent multiple-byte units from spanning across two memory words, thus allowing individual load or store instructions to make one and only one memory access -- which makes hardware easier to build
## Data in Memory

- $=\text{BYTE}=0=$
- $=\text{BYTE}=1=$
- $=\text{BYTE}=2=$
- $=\text{BYTE}=3=$

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

<<<<< one trip across the bus >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

<table>
<thead>
<tr>
<th>memory word</th>
<th>0:</th>
<th>=BYTE=0=</th>
<th>=BYTE=1=</th>
<th>=BYTE=2=</th>
<th>=BYTE=3=</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 4:</td>
<td></td>
<td>byte 4</td>
<td>byte 5</td>
<td>byte 6</td>
<td>byte 7</td>
</tr>
<tr>
<td>word 8:</td>
<td></td>
<td>byte 8</td>
<td>byte 9</td>
<td>byte a</td>
<td>byte b</td>
</tr>
</tbody>
</table>

...
### Data in Memory

**example: unaligned load/store of word at address 2**

<table>
<thead>
<tr>
<th>CPU register</th>
<th>=BYTE=2=</th>
<th>=BYTE=3=</th>
<th>=BYTE=4=</th>
<th>=BYTE=5=</th>
</tr>
</thead>
</table>

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n```

`<<<<<<<<<<<two trips across the bus>>>>>>>>>>>>>>>>>>`

<table>
<thead>
<tr>
<th>memory word 0:</th>
<th>byte 0</th>
<th>byte 1</th>
<th>=BYTE=2=</th>
<th>=BYTE=3=</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 4:</td>
<td>=BYTE=4=</td>
<td>=BYTE=5=</td>
<td>byte 6</td>
<td>byte 7</td>
</tr>
<tr>
<td>word 8:</td>
<td>byte 8</td>
<td>byte 9</td>
<td>byte a</td>
<td>byte b</td>
</tr>
</tbody>
</table>

...
Data in Memory

unaligned access:

• may be illegal (system may print "Bus error (core dumped)")
• may cause a trap into the operating system, or,
• requires extra shifting network hardware and extra time to perform two memory accesses and the necessary insertion of bytes (i.e., to merge bytes from two memory words into the single-word-length CPU register on a load, or to distribute the bytes from a register into two different memory words on a store)
Data in Memory

Alignment rules:

- bytes can start anywhere
- halfwords start on a halfword boundary = address divisible by 2
- words start on a word boundary = address divisible by 4
- doublewords start on a doubleword boundary = address divisible by 8
Data in Memory

Alignment:

- to avoid unaligned accesses the compiler can sometimes reorder or pad data structures
- You can use the .align pseudo-op in your programs (esp. when you have a word word variable allocated after a character string)

```asm
.asciz "...
.align 4
.word ...
```

(or defensive programming practice
will always allocate doublewords first,
then words, then halfwords, then strings)
Data in Memory

Alignment:

• However, in some languages (e.g., FORTRAN, COBOL), unaligned accesses must still be legal for programs, so the operating system on an aligned memory processor must include an unaligned trap handler (or the compiler must generate extra code).

• Most processors today have added hardware to support unaligned operands
Data in Memory

• Depending on which computing system you use, you will have to consider the byte order in which multibyte numbers are stored, particularly when writing those numbers to a file.
• The two orders are called "Little Endian" and "Big Endian".
• Endianness is important as a low-level attribute of a particular data format.
• Failure to account for varying endianness across architectures when writing software code for mixed platforms and when exchanging certain types of data might lead to failures and bugs.
• These issues have been understood and properly handled for many decades.
Data in Memory

Byte ordering: (pp. 71 – 72 in the 1st ed., 91 – 92 in the 2nd)
- how are bytes within a multiple byte structure ordered?
- big-endian => byte 0 is most significant = left-to-right

<table>
<thead>
<tr>
<th>word 0:</th>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 4:</td>
<td>byte 4</td>
<td>byte 5</td>
<td>byte 6</td>
<td>byte 7</td>
</tr>
</tbody>
</table>

... 

- little-endian => byte 0 is least significant = positional representation

<table>
<thead>
<tr>
<th>word 0:</th>
<th>byte 7</th>
<th>byte 6</th>
<th>byte 5</th>
<th>byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 4:</td>
<td>byte 3</td>
<td>byte 2</td>
<td>byte 1</td>
<td>byte 0</td>
</tr>
</tbody>
</table>
Byte ordering (cont’d):

- If a big-endian computer sends word 0x89abcdef as bytes 0x89, 0xab, ..., then, without conversion, a little-endian computer will receive these bytes and assemble them as 0xefcdab89

Example: 16-bit halfword with value 0xabcd at address 0x100

<table>
<thead>
<tr>
<th>little-endian</th>
<th>big-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd ab</td>
<td>ab cd</td>
</tr>
<tr>
<td>100 101</td>
<td>100 101</td>
</tr>
</tbody>
</table>
Data in Memory

Byte numbering (cont’d):

Example: 32-bit word with value 0x1234 at address 0x100

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
</tr>
</thead>
<tbody>
<tr>
<td>little-endian</td>
<td>34</td>
<td>12</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>big-endian</td>
<td>00</td>
<td>00</td>
<td>12</td>
<td>34</td>
</tr>
</tbody>
</table>
Byte numbering (cont’d):

Example: character string “abcdef” in C (null-terminated)

<table>
<thead>
<tr>
<th></th>
<th>little-endian</th>
<th>big-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100 101 102 103 104 105 106</td>
<td>100 101 102 103 104 105 106</td>
</tr>
<tr>
<td></td>
<td>‘a’ ‘b’ ‘c’ ‘d’ ‘e’ ‘f’ 0</td>
<td>‘a’ ‘b’ ‘c’ ‘d’ ‘e’ ‘f’ 0</td>
</tr>
</tbody>
</table>

There is no difference in addresses for a character string between big-endian and little-endian.
Example generated code for unaligned access within packed struct

ARM –

.file "align.c"

section .rodata

.align 2

.LC0:

.ascii "%p %p\012\000"

.text

.align 2

.global main

.type main, %function

main:

@ args = 0, pretend = 0, frame = 0
@ frame_needed = 1, uses_anonymous_args = 0

push {r7, lr}

add r7, sp, #0

movw r3, #:lower16:.LC0

movt r3, #:upper16:.LC0
Example generated code for unaligned access within packed struct

ARM –

    mov    r0, r3
    movw   r1, #:lower16:test
    movt   r1, #:upper16:test
    ldr    r2, .L2
    bl     printf
    movw   r3, #:lower16:test
    movt   r3, #:upper16:test
    ldr    r2, [r3, #4]
    uxtb   r2, r2
    orr    r2, r2, #256
    str    r2, [r3, #4]
    mov    r2, #0
Example generated code for unaligned access within packed struct

ARM -

```
strb    r2, [r3, #8]
mov     r3, #0
mov     r0, r3
pop     {r7, pc}
.L3:
    .align  2
.L2:
    .word   test+5
    .size   main, .-_main
.ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
```
Example generated code for unaligned access within packed struct

SPARC - uses multiple load/store bytes to accomplish access to unaligned word

```
.file "align.c"
.section    ".rodata"
.align 8

.LLC0:
.asciz "%p %p\n"
.section    ".text"
.align 4
.global main
.type main, #function
.proc 04

main:
  !#PROLOGUE# 0
  save   %sp, -112, %sp
```
Example generated code for unaligned access within packed struct

SPARC -

!#PROLOGUE# 1
sethi %hi(.LLC0), %g1
or %g1, %lo(.LLC0), %o0
sethi %hi(test), %g1
or %g1, %lo(test), %o1
sethi %hi(test+5), %g1
or %g1, %lo(test+5), %o2
call printf, 0
nop
sethi %hi(test), %g1
or %g1, %lo(test), %o5
ldub [%o5+5], %g1 // start of unaligned access
and %g1, 0, %g1 //
stb %g1, [%o5+5] //
ldub [%o5+6], %g1 // accesses individual bytes, and
%g1, 0, %g1 // so four ldub/stb pairs
stb %g1, [%o5+6] // instead of a single st
Example generated code for unaligned access within packed struct

SPARC -

    ldub [%o5+7], %g1 //
    and %g1, 0, %g1 //
    stb %g1, [%o5+7] //
    ldub [%o5+8], %g1 //
    and %g1, 0, %g1 //
    or %g1, 1, %g1 //
    stb %g1, [%o5+8] // end of unaligned access
    mov 0, %g1
    mov %g1, %i0
    ret
    restore

.size main, .-main
.common test,9,1 // 9-byte struct, 1-byte // boundary
.ident "GCC: (GNU) 3.4.1"
Example generated code for unaligned access within packed struct

x86 - uses single movl to access unaligned word

```
    .file "align.c"
    .version "01.01"
    .section .rodata

.LC0:
    .string "\%p \%p\n"
    .text
    .globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    addl $15, %eax
    addl $15, %eax
```
Example generated code for unaligned access within packed struct

x86 –

```assembly
shrl $4, %eax
sall $4, %eax
subl %eax, %esp
subl $4, %esp
pushl $test+5
pushl $test
pushl $.LC0
call printf
addl $16, %esp
movl $1, test+5 // unaligned access
movl $0, %eax
leave
ret
.size main, .-main
.comm test, 9, 1
.ident "GCC: (GNU) 3.4.1"
```