Endianness

Systems Software

Addresses of multi-byte objects

- Ex: consider the declaration

```
int x;
```

  on a 32-bit machine. `x` takes up 4 bytes in memory.

- So if `x` occupies the addresses 0x100 through 0x103, what is the address of x (value of the C expression `&x`) ?
- Convention: the address of a multi-byte object is the smallest of the addresses used to store the bytes.

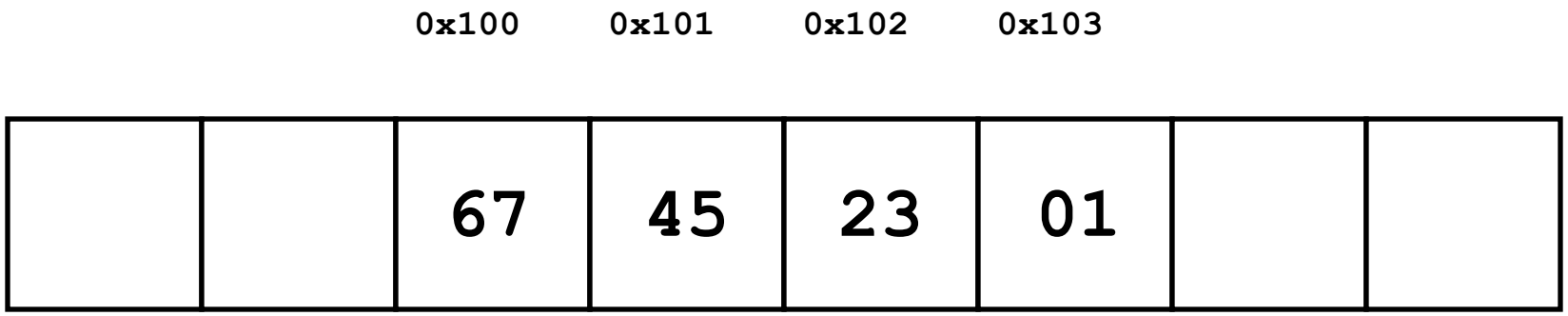
Byte Ordering

- **Two popular conventions**
- **Intel is a little endian architecture**
  - Least significant byte is stored at lowest address
- **Some other systems use big endian**
  - Most significant byte is stored at lowest address

- **Ex: How would int x = 0x01234567 at address 0x100 be stored?**

Little Endian

- Ex: How would int x = 0x01234567 at address 0x100 be stored?



Origin of Big & Little Endians

Jonathan Swift, *Gulliver's Travels*

- **A law requiring all citizens of Lilliput to break their soft-eggs at the little ends only**
- **A civil war breaking between the Little Endians and the Big Endians, resulting in the Big Endians taking refuge on a nearby island, the kingdom of Blefuscu**
- **Satire over holy wars between Protestant Church of England and the Catholic Church of France**

Do we care?

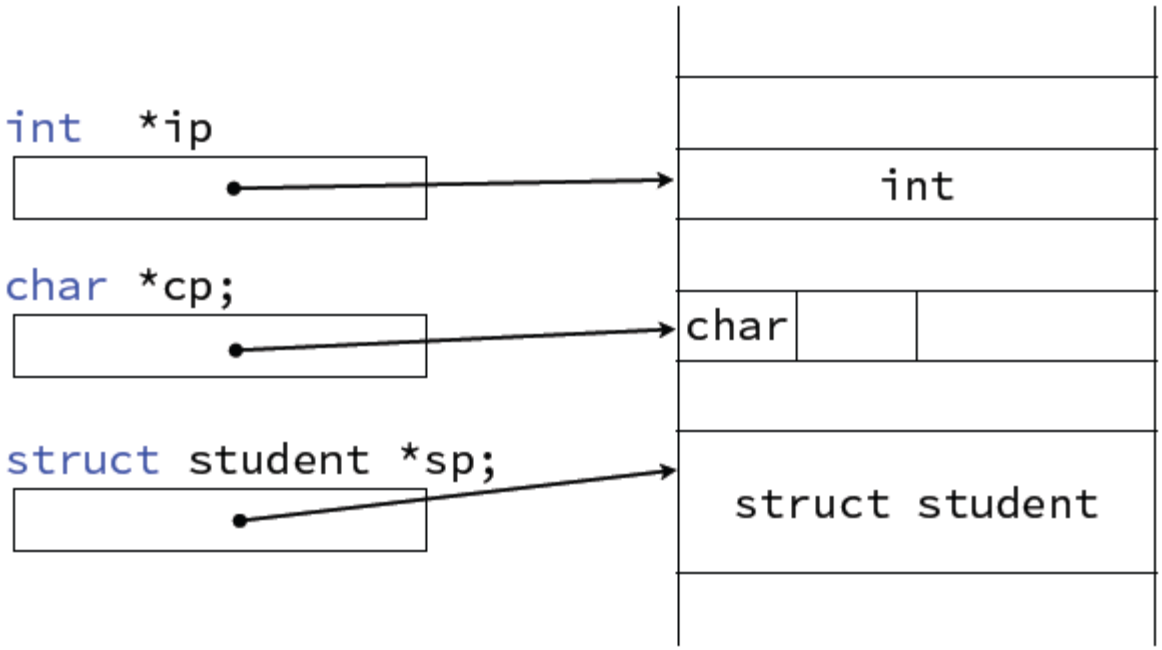
- **For most programs we don't have to worry about endianness … variables generally work as you would expect**

- **Endianness matters**
  - When transferring data over a network between machines with different endianness
  - When inspecting binary data representing integers. Consider the following IA32 instruction and its disassembly:

```
80483bd:    01  05  64  94  04  08        add   %eax,   0x08049464
```

Pointers

- A pointer is a variable declared to store a memory address
- declaration syntax: type *var_name;
- Pointers can be to any data type, including structures



Pointers

- **Important pointer-related operators**

  & : address-of

  *** : dereference (*not the same as the * used for declarations!!!*)**

- **Example**

```
int i = 5;  /* i is an int containing 5 */
int *p;     /* p is a pointer to an int */
int j;      /* j is an uninitialized int */

p = &i;     /* store the address of i in p */
j = *p;     /* store the value p points to into j */
```

Pointers and Arrays

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

```
int a[10];
int *ptr;

/* following statements are same */
ptr = a;
ptr = &a[0];
```

The sizeof operator

- **The sizeof operator gets a variable or a type as an input and outputs its size in bytes.**
- **Typical sizes on a 32-bit machine**

```
sizeof(char)   == 1
sizeof(short)  == 2
sizeof(int)    == 4
sizeof(int *)  == 4
```

- **Lookup**
  - signed/unsigned – int, char, short

Pointer arithmetic

- Addition/subtraction operations on pointers work in multiples of the size of the data type being pointed to
- Example

```
char   *cp1;
int    *ip1;
double *dp1;

cp1++;          /* Increments address by 1 */
ip1++;          /* Increments address by 4 */
dp1++;          /* Increments address by 8 */
```

Casting

- **Reference an object according to a different data type from which it was created.**
- **You can cast pointers to be pointers to other types. This is powerful.**

Code to print byte representation

```
int main(void) {
  int i=12345, j;
  unsigned char *p = (unsigned char *) &i;

  for (j=0; j<4; j++)
    printf("Offset %d: %x\n", j, p[j]);
}
```

Output on a little-endian machine

Offset 0: 39
Offset 1: 30
Offset 2: 0
Offset 3: 0

Decimal:	12345
Binary:	0011 0000 0011 1001
Hex:	3 0 3 9

show_bytes routine P. 37

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len){
  int i;
  for (i = 0; i < len; i++)
    printf(" %.2x", start[i]);
  printf("\n");
}
```

Problem P. 40

Consider the following three calls to show_bytes

```
int val = 0x12345678;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1);  /*  A. */
show_bytes(valp, 2);  /*  B. */
show_bytes(valp, 3);  /*  C. */
```

Indicate the values that would be printed by each call on a little-endian machine and on a big-endian machine.

Problem P. 100

- Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should always run on any machine, regardless of its word size.