CS 4400
Computer Systems

LECTURE 2

Information storage

Bit-level operations

New to C?

What does the following bit pattern represent?

1000 1000 1000 1000 0001 0001 0001 0001

A. an unsigned integer > 231

B. a negative integer

C. a normalized floating-point value

D. four characters

E. an x86 assembly-language instruction

F. none of the above

G. all of the above

Bits

- All information stored by computers reduces to groups of two-valued signals, ***bits***.
- Only when we apply some ***interpretation*** to the different possible bit patterns does a group of bits have meaning.
- Three important encodings
  - unsigned integers: $x \geq 0$
  - two's complement integers: x may be positive, negative, or 0
  - floating-point numbers: approximate real values
- We can represent the values of any ***finite*** set.

Limitations

- Due to using a limited number of bits to encode a value, overflow (or underflow) can occur.

```
int x = 1000000000;
int y = 2000000000;

int z = x + y;  // z is -1294967296
```

- Computer arithmetic does not follow every rule of integer arithmetic.
  - ✕ The sum of two positive integers is a positive integer.
- However, computer arithmetic is consistent.
- Ariane 5 explosion

Why Do We Care?

- By understanding
  - the ranges of values that can be represented
  - the properties of arithmetic operations
- We can write programs that
  - work correctly over the full range of values
  - are portable across different machines and compilers
- Learning how to implement arithmetic operations by ***directly manipulating the bits that represent numbers*** is critical to understanding the machine-level code generated.

Addressing Bytes

- Bits are accessible in 8-bit blocks, ***bytes***.
- To a machine-level program, memory is simply a very large array of bytes, virtual memory
- A unique number identifies each such byte, ***virtual memory address***
- The set of all possible addresses, the ***virtual memory address space***, is merely conceptual
- The mapping of virtual memory addresses to physical (i.e., real) addresses will be covered later

Binary Notation

- Each binary digit has a position ***p***, starting with the least-significant bit (LSB) at ***p = 0*** and proceeding to the most-significant bit (MSB) at ***p = bitCount - 1***.
- Written with LSB on the right and MSB on the left.
- If the bit at position p is 1, it contributes 2p to the decimal value of the number being represented.

  $x = bit_{bitCount - 1} * 2^{bitCount - 1} + ... + bit_1 * 2^1 + bit_0 * 2^0$
- Decimal value 23 in binary notation?

Hexadecimal Notation

- Base 16, using digits 0-9 and characters A-F to represent the 16 possible values.
- Easiest to convert from binary in 4-bit groups.
- In C, numeric constants starting with 0x or 0X are interpreted as being in hexadecimal.
- Decimal value 23 in hex?
- Binary value 10011100 in hex?

hex	*decimal*	*binary*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Conversions

- See **`decimal_to_hex.c`**
- See **`hex_to_decimal.c`**
- See **`binary_conversions.c`**

(Sample code is uploaded to the Canvas page for CS 4400, look under the “files” tab at the left side of the page, then in the folder for Lecture 2)

Words

- Every computer has a ***word size***, which indicates the size of integer and pointer data.
- How does the word size determine the maximum of the virtual address space?
- For a machine with an ***n-bit*** word size, virtual addresses can range from 0 to 2^n-1.
- For computers that are 32-bit, the virtual address space is limited to 4 GB.
- What's the limit for 64-bit?

Data Sizes

- Computers and compilers support multiple data formats in different lengths.
- C supports data formats for both integers and floating-point.

	typical 32-bit	typical 64-bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`char*`	4	8
`float`	4	4
`double`	8	8
`quadruple`	16	16

Portability

- One aspect of portability is to make programs insensitive to the exact sizes of different data types.
- Because 32-bit machines have been the standard for so long, older programs assume the “typical 32-bit” sizes.
- With the increasing prominence of 64-bit machines, hidden word dependences have surfaced as bugs.
- For example, using an `int` to store a pointer can be problematic.

Addressing Multi-Byte Data

- For an object that spans multiple bytes, we must consider
  - how to address the object
  - how the bytes are ordered
- The object's address is that of the smallest of the bytes.
- For example, an `int` stored in four bytes at memory locations `0x100`, `0x101`, `0x102`, and `0x103` has address `0x100`.

Two Byte Ordering Conventions

- Consider a *w*-bit integer with bit representation

  $x_{w-1}\ x_{w-2}\ \ldots\ x_1\ \ x_0$ with MSB x_{w-1} and LSB x_0
- Assume *w* is a multiple of 8, to group the bits in bytes.
- The most-significant byte has bits $x_{w-1}\ x_{w-2}\ \ldots\ x_{w-7}\ x_{w-8}$
- The least-significant byte has bits $x_7\ x_6\ \ldots\ x_1\ x_0$
- *Little endian*—the least-significant byte comes first
- *Big endian*—the most-significant byte comes first

Example: Byte Order

- Little endian:

  $x_7\ x_6\ \ldots\ x_1\ x_0 \mid x_{15}\ x_{14}\ \ldots\ x_9\ x_8 \mid x_{23}\ x_{22}\ \ldots\ x_{17}\ x_{16} \mid \ldots$

- Big endian:

  $\ldots \mid x_{23}\ x_{22}\ \ldots\ x_{17}\ x_{16} \mid x_{15}\ x_{14}\ \ldots\ x_9\ x_8 \mid x_7\ x_6\ \ldots\ x_1\ x_0$

- Consider

```
int x = 0x01234567;  // 19088743
int* addr = &x;      // 0x100
```

	…	0x100	0x101	0x102	0x103	…
?? endian	…	01	23	45	67	…
?? endian	…	67	45	23	01	…

- When is byte order an issue for the programmer?

Representing Strings

- In C, a string is an array of characters terminated with a special character ‘`\0`’ (the null character, value `0x0`).
- Each character is simply an integer code (usually ASCII).
- Example 1: “`hello`”

  `68 65 6C 6C 6F 00`

- Example 2: “`1234567`”

  `31 32 33 34 35 36 37 00`

- These examples are independent of byte ordering and word size. Why?
- How are Java strings different?

Representing Code

- From the perspective of the machine, a program is simply a sequence of bytes.
- Example:

```
int sum(int x, int y) {
  return x + y;
}
```

  - Linux `05 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`
  - Sun `81 c3 e0 08 90 02 00 09`
- Binary code is seldom portable across different machines.

Question

- Suppose that

```c
int x = 0xAA;
int y = 0x55;
```

- What are the results of the following C expressions?

<table>
<tr><td>And:</td><td colspan="6"><code>x & y</code></td><td colspan="6"><code>x && y</code></td></tr>
<tr><td>(X)Or:</td><td colspan="4"><code>x | y</code></td><td colspan="4"><code>x ^ y</code></td><td colspan="4"><code>x || y</code></td></tr>
<tr><td>Not:</td><td colspan="3"><code>~x</code></td><td colspan="3"><code>~y</code></td><td colspan="3"><code>!x</code></td><td colspan="3"><code>!y</code></td></tr>
<tr><td>Shift:</td><td colspan="3"><code>x<<2</code></td><td colspan="3"><code>y<<2</code></td><td colspan="3"><code>x>>2</code></td><td colspan="3"><code>y>>2</code></td></tr>
</table>

Operations in C

- See `bit_level_ops.c`
- See `logical_ops.c`
  - Be careful not to confuse bit-level and logical ops.
  - *What is short-circuit evaluation?*
- See `shift_ops.c`
  - Left shift always fills with 0s.
  - Right shift may be logical (fills w/0s) or arithmetic (fills w/value of MSB).

New to C? – Pointers

- You are already familiar with accessing variables using their names (same as in Java).

```
int num = 10;
```

- We can also access `num` through a second variable that holds the address of variable `num`.
- The pointer variable `ptr` holds the address of `num`.

```
int* ptr = &num;
```

- `&` immediately to the left of a variable gives an expression whose value is the variable's virtual memory address.

Pointers and Addresses

- Suppose the address of num is 0x9640.
- Variable `ptr` “points to” `num`: `ptr = #`

- To access the contents of a cell whose addresses is in `ptr`, dereference the pointer using `*ptr`.
`*ptr = 3;`

Declaring Pointers

- To declare `ptr` as a pointer variable that can hold the address of an `int` variable:

```
int* ptr;
```

- The data type is `int*`, the variable is `ptr`.
- Be careful when declaring multiple variables on the same line. In

```
int* ptr1, ptr2;
```

- The variable `ptr2` is a regular `int`.
- To declare two pointers:

```
int *ptr1, *ptr2;
```

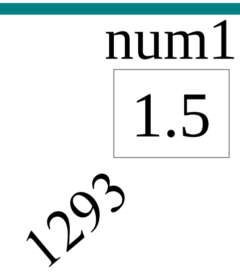
Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `float temp;`
4. `float* flt_ptr;`
5. `flt_ptr = &num1;`
6. `temp = *flt_ptr;`
7. `*flt_ptr = num2;`
8. `num2 = temp;`

What values do you expect to be contained in `num1`, `num2`, `temp`, and `flt_ptr`?

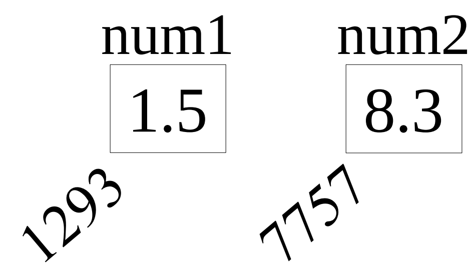
Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `float temp;`
4. `float* flt_ptr;`
5. `flt_ptr = &num1;`
6. `temp = *flt_ptr;`
7. `*flt_ptr = num2;`
8. `num2 = temp;`



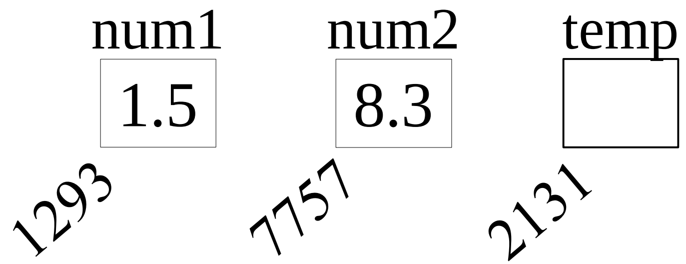
Example: Swapping Variables

```
float num1 = 1.5;
float num2 = 8.3;
float temp;
float* flt_ptr;
flt_ptr = &num1;
temp = *flt_ptr;
*flt_ptr = num2;
num2 = temp;
```



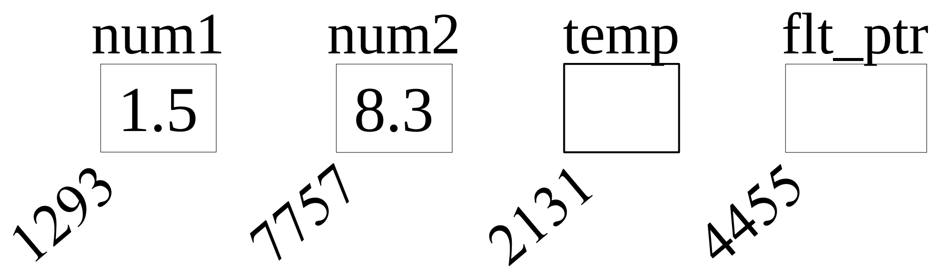
Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `float temp;`
4. `float* flt_ptr;`
5. `flt_ptr = &num1;`
6. `temp = *flt_ptr;`
7. `*flt_ptr = num2;`
8. `num2 = temp;`



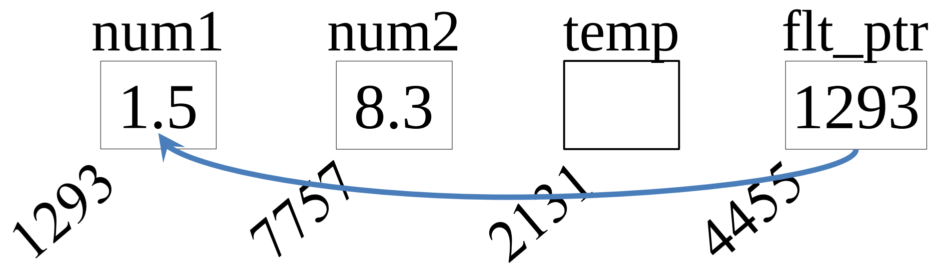
Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `float temp;`
4. `float* flt_ptr;`
5. `flt_ptr = &num1;`
6. `temp = *flt_ptr;`
7. `*flt_ptr = num2;`
8. `num2 = temp;`



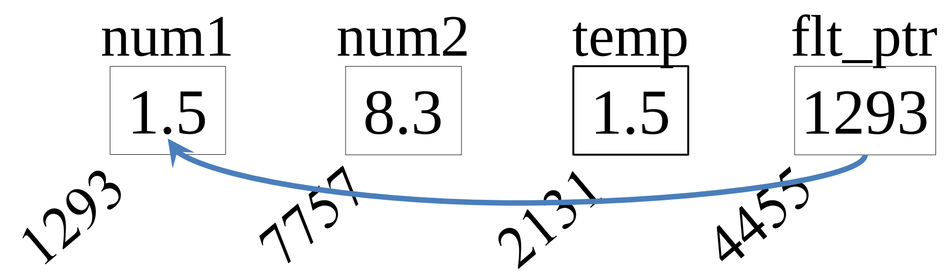
Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `float temp;`
4. `float* flt_ptr;`
5. `flt_ptr = &num1;`
6. `temp = *flt_ptr;`
7. `*flt_ptr = num2;`
8. `num2 = temp;`



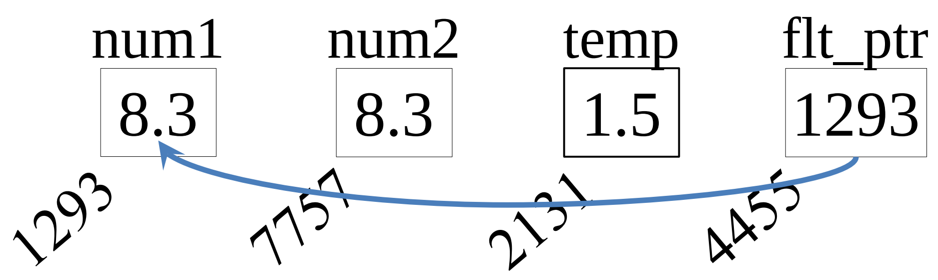
Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `float temp;`
4. `float* flt_ptr;`
5. `flt_ptr = &num1;`
6. `temp = *flt_ptr;`
7. `*flt_ptr = num2;`
8. `num2 = temp;`



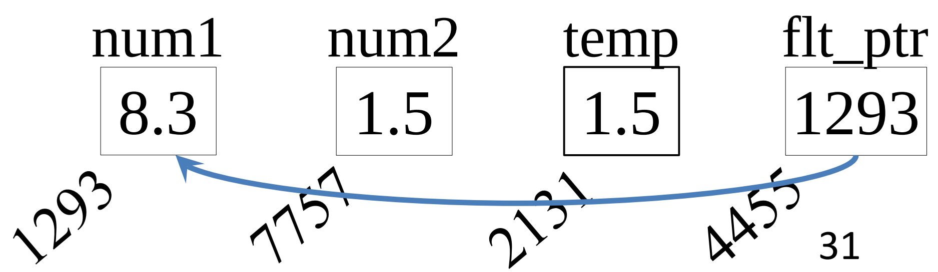
Example: Swapping Variables

```
float num1 = 1.5;
float num2 = 8.3;
float temp;
float* flt_ptr;
flt_ptr = &num1;
temp = *flt_ptr;
*flt_ptr = num2;
num2 = temp;
```



Example: Swapping Variables

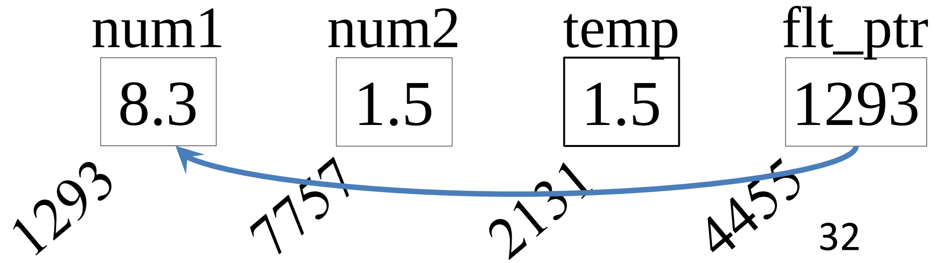
```
float num1 = 1.5;
float num2 = 8.3;
float temp;
float* flt_ptr;
flt_ptr = &num1;
temp = *flt_ptr;
*flt_ptr = num2;
num2 = temp;
```



Example: Swapping Variables

1. `float num1 = 1.5;`
2. `float num2 = 8.3;`
3. `flo`
4. `flo`
5. `flt`
6. `tem`
7. `*flt_ptr = num2;`
8. `num2 = temp;`

Why do we have pointers? It seems like a more complicated way to do something we could already do!



Pointers and Arrays

- An array name is a pointer constant whose value is the address of the first array element, and the value cannot be changed.
- A pointer variable has a value that is an address, and it can be changed.
- Example:

```
float rates[100];
float *ptr;
ptr = rates;  /* needs no & */
```

- Last line equivalent to ptr = &rates[0]; .

Dynamically-Allocated Arrays

- How do you deal with an array when you don't know at compile time how large it should be?

```
int my_array[100000];  //big enough?
```

- Allocate memory at run time, using library routine malloc.

```
int x = count_of_bytes_given_by_user;
int* my_array = malloc(x);
// my_array is address of first element
// my_array+1 is address of second
```

- Much more on dynamic memory allocation to come.

Pointers and Strings

- Recall that strings are really `char` arrays.

```
char my_string[] = “hello”;
```

- We can have a pointer to the array.

```
char *ptr = my_string;
```

- In fact, we can directly initialize the pointer with the string.

```
char *ptr = “hello”;
```

- What is the difference between `ptr` and `my_string`?

Pointer Arithmetic

- Pointer arithmetic can access individual array elements.
- Ops `++` and `--` increment/decrement pointers.
- The result of incrementing a pointer is that it points to the next cell in the array (works regardless of the data size).
- Other operations may be applied to pointers (`+`, `-`, `<`, `>`).
- Example:

```
float nums[] = { 1.2, 3.4, 5.6 };
float *p1 = nums;
float *p2 = p1 + 2;
```

- Value of `*p2`? Is expression `p1 < p2` true or false?

Exercise: Pointers

- Write a function `check` with two parameters: `char* str` and `char c`.
- Function `check` returns `1` if `c` is in `str` and `0` otherwise.

  (See `check.c`)

New to C? – Formatted Output

- Function `printf` performs formatted output, in that it
  - controls where data is written
  - converts input into the desired type
  - writes output in the desired manner
- To print to standard out, use `printf(format_str, arg1, ..., argN)`
- Functions for printing to file and to string also exist, and are similar (`fprintf` and `sprintf`, respectively).
- Example:

```
printf("%i%c%i is %f", 1, '/', 2, 0.5);
```

Format String and Address List

- The `format_str` and argument list (`arg1,...,argN`) should correspond.
- An item in the `format_str` specifies how the argument should be converted for output.
- The matching item in the argument list specifies what value should be printed. This list may contain any valid C expression, even function calls.
- The format string may contain any ordinary characters and conversion codes (denoting how to convert output).

Conversion Codes

- `%d`, `%i` decimal number
- `%x`, `%X` unsigned hexadecimal number
- `%c` single character
- `%s` characters from string until reaching `'\0'`
- `%f` floating-point number (default precision: 6)

- Note: see K&R for more conversion codes and options (field width, max chars/digits printed, alignment, ...).

New to C? – Casting

- In C, it is possible to explicitly convert one data type to another (pointer types included).
- For example, suppose that `x` is of type `int`. The expression `(float)x` is the original value of `x` converted to `float`.
- Note that the actual value and type of `x` are unchanged.
- Casting may also be implicit. In mixed-type expressions, the types of some values are (invisibly) changed.

Example: Casting

```
#include <stdio.h>

int main(void) {
  int miles;
  int hours;
  float mph;

  miles = 455;
      hours = 3;

      mph = miles / hours;
  printf(“%f\n”, mph);

  mph = (float) miles / (float) hours;
                    printf(“%f\n”, mph);

  return 0;
}
```

```
unix> gcc casting.c
unix>./a.out
151.000000
151.666672
```

casting.c

Mixed-Mode Arithmetic

- When variables of different types are included in a single arithmetic expression, the values are converted to the same type before the operation is performed.
- For example, the value of `int` variable `x` is converted to type float before the division is performed.

```
x / 4.0
```

- Again, the actual type and value of `x` are unchanged.
- Conversion to the same, more general type. E.g., converts `int` to `float`, not `float` to `int`.

Type Promotion Hierarchy

- Types are organized into a promotion hierarchy.
  - `long double`
  - `double`
  - `float`
  - `unsigned long`
  - `long`
  - `unsigned int`
  - `int`
  - `unsigned short`
  - `short`
  - `unsigned char`
  - `char`

more general

Example: Mixed-Mode Arithmetic

- Pay attention to when the type conversion occurs.
- Notice difference in implicit and explicit conversion.
- Example:

```
float a, b;
int c, d;

b = 1.0;
c = -5;
d = 2;

a = b * (c / d);              /* a is -2.0 */
a = b * ((float)c / d);       /* a is -2.5 */
a = b / c * d;                /* a is -0.4 */
a = (int)(b / c) * d;         /* a is 0.0 */
```