Study Group

- Mondays and Wednesdays 11:00 AM – 1:00PM
- Where: Undergraduate lounge near the CS off ce in the MEB building.
- Questions contact Zach Lewis via e-mail: Gonzoga56@gmail.com
- Even if you cannot make it for the whole time, still feel free to stop by when you're free.

CS 4400
Computer Systems

LECTURE 2

Information storage

Bit-level operations

New to C?

Clicker Question

If you have ResponseCard clicker, channel is **41**.

If you are using ResponseWare, session id is **CS1400U**.

What does the following bit pattern represent?

1000 1000 1000 1000 0001 0001 0001 0001

A. an unsigned integer > 2^{31}

B. a negative integer

C. a normalized floating-point value

D. four characters

E. an x86 assembly-language instruction

F. I don't know

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.

Why Do We Care?

- By understanding
  - the ranges of values that can be represented and
  - the properties of arithmetic operations,

  we can write programs that
  - work correctly over the full range of values and
  - 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 sophisticated 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 $p = \text{bitCount} - 1$.
- Written with LSB on the right and MSB on the left.
- If the bit at position *p* is 1, it contributes 2^p to the decimal value of the number being represented.
- $x = \text{bit}_{\text{bitCount} - 1} * 2^{\text{bitCount} - 1} + \ldots + \text{bit}_1 * 2^1 + \text{bit}_0 * 2^0$
- 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`

(All sample code is provided on the class website.)

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-pt.

	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

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 and
  - 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

  $x_{w\text{-}1}\ x_{w\text{-}2} \ldots x_1\ x_0$ with MSB $x_{w\text{-}1}$ and LSB x_0

- Assume *w* is a multiple of 8, to group the bits in bytes.
- The most-significant byte has bits $x_{w\text{-}1}\ x_{w\text{-}2} \ldots x_{w\text{-}7}\ x_{w\text{-}8}$.
- 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: $x_7\ x_6 \ldots x_1\ x_0\ \ x_{15}\ x_{14} \ldots x_9\ x_8\ \ x_{23}\ x_{22} \ldots x_{17}\ x_{16} \ldots$
- Big endian: $\ldots x_{23}\ x_{22} \ldots x_{17}\ x_{16}\ x_{15}\ x_{14} \ldots x_9\ x_8\ \ x_7\ x_6 \ldots x_1\ x_0$
- 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?

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.

Clicker Question

Suppose that

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

What is the result of the following C expression?

```
x & y
```

A. `0`

B. `1`

C. `0x11`

D. `0xFF`

E. I don't know

Clicker Question

Suppose that

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

What is the result of the following C expression?

```
x || y
```

A. `0`

B. `1`

C. `1`, only the value of x is considered

D. `0xFF`

E. I don't know

Boolean Algebra

By encoding values True and False as 1 and 0, Boolean algebra captures the properties of prepositional logic.

$\neg$	
0	1
1	0

(NOT, ~ in C)

$\wedge$	0	1
0	0	0
1	0	1

(AND, & in C)

$\vee$	0	1
0	0	1
1	1	1

(OR, | in C)

$\oplus$	0	1
0	0	1
1	1	0

(XOR, ^ in C)

Boolean Algebra Properties (1)

- Commutativity `a | b = b | a` `a & b = b & a`
- Associativity `(a | b) | c = a | (b | c)`
  `(a & b) & c = a & (b & c)`
- Distributivity `a & (b | c) = (a & b) | (a & c)`
  `a | (b & c) = (a | b) & (a | c)`
- Identity `a | 0 = a` `a & 1 = a`
- Annihilator (maps to zero) `a & 0 = 0`
- Cancellation `~(~a) = a`

Boolean Algebra Properties (2)

- Complement `a | ~a = 1` `a & ~a = 0`
- Idempotency `a & a = a` `a | a = a`
- Absorption `a | (a & b) = a`
  `a & (a | b) = a`
- DeMorgan's laws `~(a & b) = ~a | ~b`
  `~(a | b) = ~a & ~b`

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

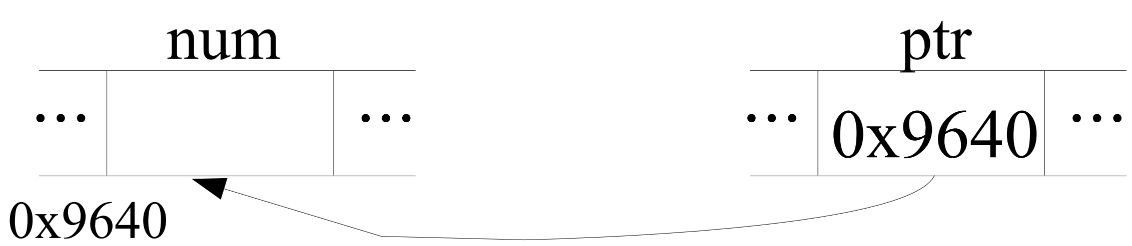
- You are already familiar with accessing variables using their names (same as in Java). `int num = 10;`
- We can also access `num` through a second variable that holds the address of variable `num`.
- The pointer variable `ptr` holds the address of `num`.

```
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`.
- `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;
```

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

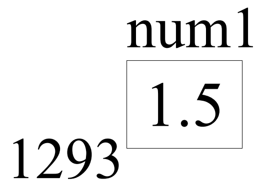
```
int *ptr1, *ptr2;
```

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;
```

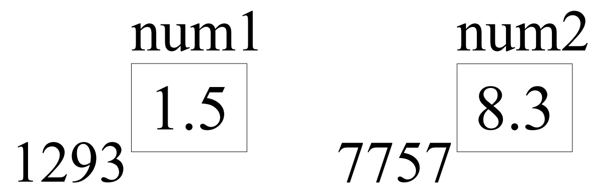
Example: Swapping Variables

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



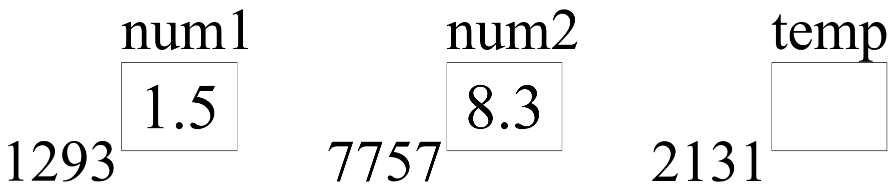
Example: Swapping Variables

```
float num1 = 1.5;

float num2 = 8.3;

float temp;

float* flt_ptr;

flt_ptr = &num1;

temp = *flt_ptr;

*flt_ptr = num2;

num2 = temp;
```



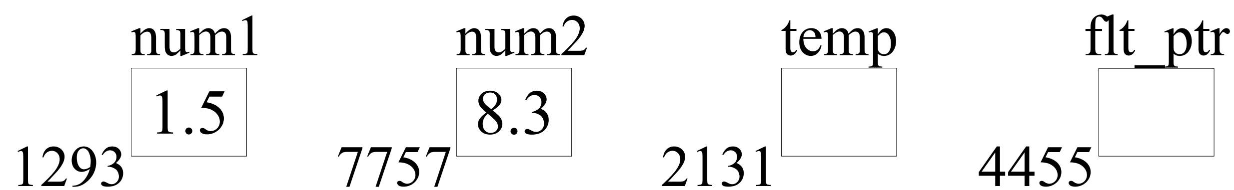
Example: Swapping Variables

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



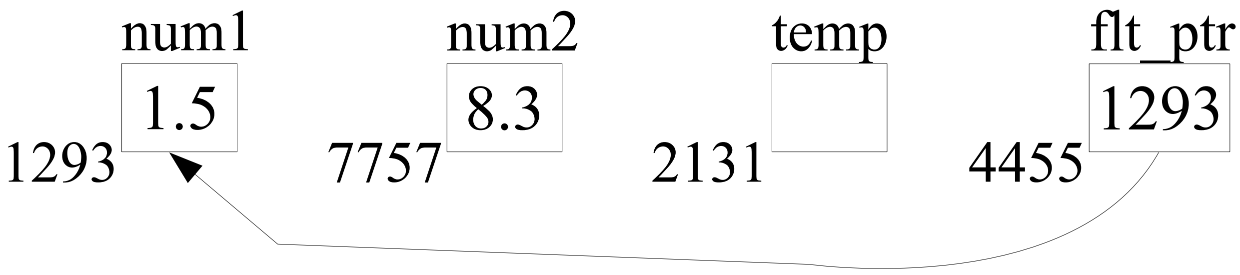
Example: Swapping Variables

```
float num1 = 1.5;

float num2 = 8.3;

float temp;

float* flt_ptr;

flt_ptr = &num1;

temp = *flt_ptr;

*flt_ptr = num2;

num2 = temp;
```



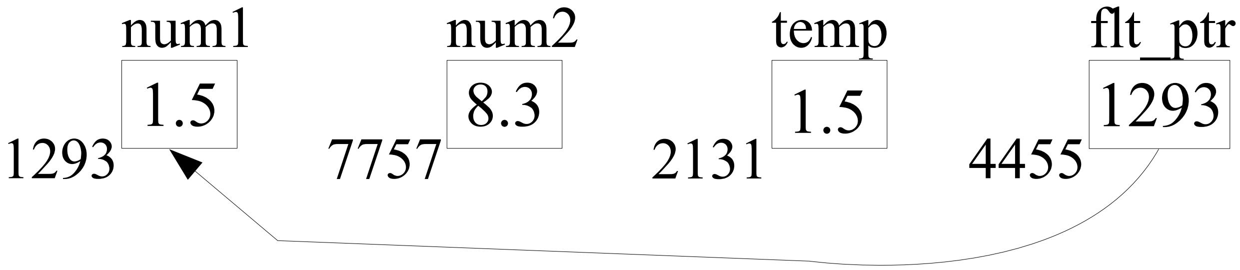
Example: Swapping Variables

```
float num1 = 1.5;

float num2 = 8.3;

float temp;

float* flt_ptr;

flt_ptr = &num1;

temp = *flt_ptr;

*flt_ptr = num2;

num2 = temp;
```



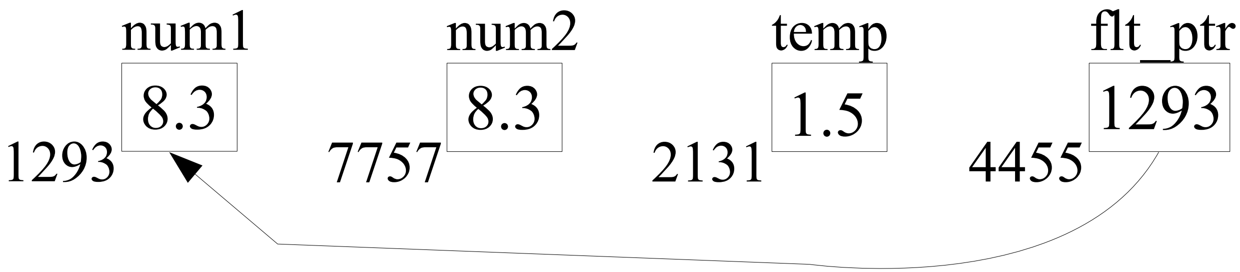
Example: Swapping Variables

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



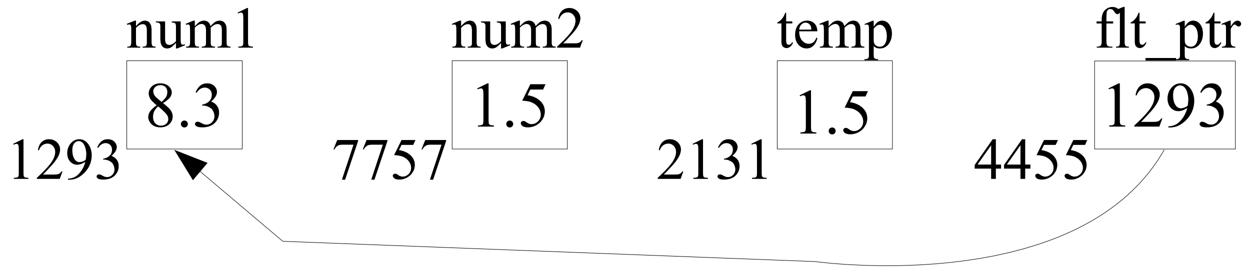
Example: Swapping Variables

```
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

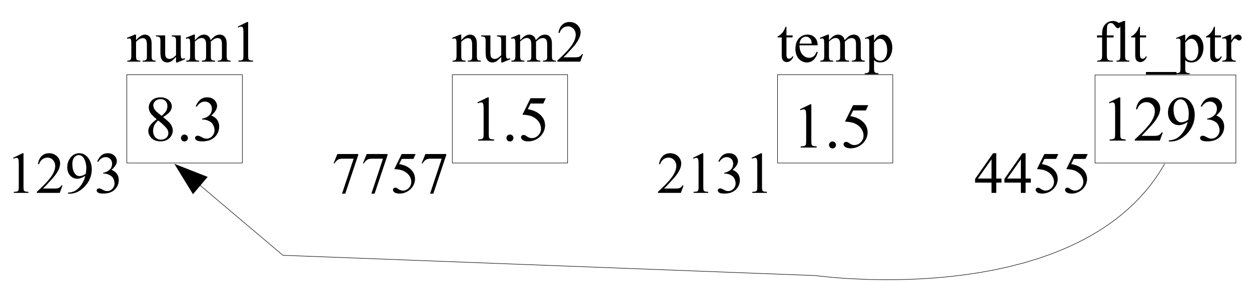
```
float num1 = 1.5;

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

float temp;

float* flt_

flt_ptr = &

temp = *flt_ptr;

*flt_ptr = num2;

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 in `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, and
  - writes output in the desired manner.
- `printf(format_str, arg1, ..., argN)` prints to standard output.
- 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

- `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)
- 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

casting.c

```
#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
```

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 */
```