CS 33 Discussion 4

April 28, 2017

Agenda

Pexex Lab

HW3

Debugging Process

Reproduce the bug
Simplify program input
Use a debugger to track down the origin of the problem
Fix the problem

GDB – GNU Debugger

Debugger for several languages

C, C++, Java, Objective-C... more

Allows you to inspect what the program is doing at a certain point during execution

Logical errors and segmentation faults are easier to find with the help of gdb

Using GDB

1. Compile Program

Normally: `$ gcc [flags] <source files> -o <output file>`

Debugging: `$ gcc [other flags] -g <source files> -o <output file>`

enables built-in debugging support

2. Specify Program to Debug

```
$ gdb <executable>
```

or

```
$ gdb
(gdb) file <executable>
```

Using GDB

3. **Run Program**

   `(gdb) run` or
   `(gdb) run [arguments]`

4. **In GDB Interactive Shell**

   Tab to Autocomplete, up-down arrows to recall history
   `help [command]` to get more info about a command

5. **Exit the gdb Debugger**

   `(gdb) quit`

Setting Breakpoints

Breakpoints

used to stop the running program at a specific point

If the program reaches that location when running, it will pause and prompt you for another command

Example:

`(gdb) break file1.c:6`

Program will pause when it reaches line 6 of file1.c

`(gdb) break my_function`

Program will pause at the first line of `my_function` every time it is called

`(gdb) break [`*`position`*`] if `*`expression`*

Program will pause at specified position only when the expression evaluates to true

Breakpoints

Setting a breakpoint and running the program will stop program where you tell it to

You can set as many breakpoints as you want

`(gdb) info breakpoints|break|br|b` shows a list of all breakpoints

Deleting, Disabling and Ignoring BPs

```
(gdb) delete [bp_number | range]
```

Deletes the specified breakpoint or range of breakpoints

```
(gdb) disable [ bp_number | range]
```

Temporarily deactivates a breakpoint or a range of breakpoints

```
(gdb) enable [ bp_number | range]
```

Restores disabled breakpoints

If no arguments are provided to the above commands, all breakpoints are affected!!

```
(gdb) ignore bp_number iterations
```

Instructs GDB to pass over a breakpoint without stopping a certain number of times.

bp_number: the number of a breakpoint

Iterations: the number of times you want it to be passed over

Displaying Data

Why would we want to interrupt execution?

- to see data of interest at run-time:
- `(gdb) print [/format] expression`
  - Prints the value of the specified expression in the specified format
- Formats:
  - d: Decimal notation (default format for integers)
  - x: Hexadecimal notation
  - o: Octal notation
  - t: Binary notation

Resuming Execution After a Break

When a program stops at a breakpoint

- 4 possible kinds of gdb operations:
  - **c or continue:** debugger will continue executing until next breakpoint
  - **s or step:** debugger will continue to next source line
  - **n or next:** debugger will continue to next source line in the current (innermost) stack frame
  - **f or finish:** debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller
    - the function's return value and the line containing the next statement are displayed

Watchpoints

Watch/observe changes to variables

```
(gdb) watch my_var
```

sets a watchpoint on my_var

the debugger will stop the program when the value of *my_var* changes

old and new values will be printed

```
(gdb) rwatch expression
```

The debugger stops the program whenever the program reads the value of any object involved in the evaluation of *expression*

Stack Info

A program is made up of one or more functions which interact by calling each other

Every time a function is called, an area of memory is set aside for it. This area of memory is called a **stack frame** and holds the following crucial info:

- storage space for all the local variables
- the memory address to return to when the called function returns
- the arguments, or parameters, of the called function

Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**

Analyzing the Stack in GDB

`(gdb) backtrace|bt`

- Shows the call trace (the call stack)
- Without function calls:
  - #0 main () at program.c:10
  - one frame on the stack, numbered 0, and it belongs to main()
- After call to function display()
  - #0 display (z=5, zptr=0xbffffb34) at program.c:15
  - #1 0x08048455 in main () at program.c:10
  - Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().
  - Each frame listing gives
    - the arguments to that function
    - the line number that's currently being executed within that frame

Analyzing the Stack

(gdb) info frame

Displays information about the current stack frame, including its return address and saved register values

(gdb) info locals

Lists the local variables of the function corresponding to the stack frame, with their current values

(gdb) info args

List the argument values of the corresponding function call

Other Useful Commands

(gdb) info functions

Lists all functions in the program

(gdb) list

Lists source code lines around the current line

gdb - Debugger

For Lab 2, you may find these lines useful:

```
(gdb) break <function_name>
(gdb) run
(gdb) stepi
(gdb) stepn
(gdb) info registers
(gdb) disassemble
All variants of "print"
```

Pexex Lab

● The objective:

– Examine the execution of an actual program with a debugger.

● You will be examining the execution of Emacs, the text editor.

● Emacs also provides an interpreter for handling “Elisp”, a functional programming language that allows for simple computation.

Pexex Lab

You need to gather information for your trace and put into the file trace.tr

The executable is on either lnxsrv07 or lnxsrv09:

~eggert/bin64/bin/emacs-25.2

The corresponding source code can be found in:

~eggert/src/emacs-25.2/

Pexex Lab

- How to multiply numbers from emacs:

– ~eggert/bin64/bin/emacs-25.2 -batch -eval '(print (* 1250284240 -1844255039))'

- To run this command with gdb:

– gdb --args ~eggert/bin64/bin/emacs-25.2 -batch -eval '(print (* 1250284240 -1844255039))'

- Or

– gdb ~eggert/bin64/bin/emacs-25.2

– (gdb) r -batch -eval '(print (* 1250284240 -1844255039))'

Pexex Lab

- After opening gdb:
- Run the program

– run (or simply 'r')

- Run the program with arguments

– r arg1 arg2

- This will run the executable to completion.

Pexex Lab

- Set breakpoint at function foo:

– break foo

- Set breakpoint at instruction address 0x100

– break *0x100

– Note the asterisk. Without it, it will look for a function called 0x100. Please don't write a function called 0x100.

- When program is run or “continued”, it will run until it hits a breakpoint in which it will stop.
- Resume a program that is stopped

– continue (or just 'c')

Pexex Lab

- Once you reach a breakpoint, you can step through each instruction.
- Execute the next instruction, stepping INTO functions:

– stepi (or just 'si')

- Execute the next line of source code, stepping INTO functions:

– step (or just 's')

Pexex Lab

- Execute the next instruction, stepping OVER functions:

– nexti (or just 'ni')

- Execute the next line of source code, stepping OVER functions:

– next (or just 'n')

Pexex Lab

- Examining the assembly instructions of function foo:

– disassemble foo (or just “disas foo”)

- If you are stopped at some instruction in “foo”, the disassembled assembly will also show you where execution is paused at:

```
      0x54352e <arith_driver+46>: 49 89 d7 mov %rdx,%r15
=>    0x543531 <arith_driver+49>: 49 89 f5 mov %rsi,%r13
      0x543534 <arith_driver+52>: 41 89 fe mov %edi,%r14d
```

- This means arith_driver+46 has been executed but arith_driver+49 has not yet been executed.

Pexex Lab

● disas /m <function name>

– Display assembly for <function name> prefaced with the corresponding lines of C.

● Using next(i) or step(i), you can also set the debugger to print each instruction as it's executed:

– set disassemble-next-line on

Details on dissassemble - https://sourceware.org/gdb/onlinedocs/gdb/Machine-Code.html

Pexex Lab

- Examining registers:
- Print the current state of all registers:

– info registers

- Print the state of a particular register:

– info registers $rdi

- Print out contents at some memory address with the “x” command.

– x [addr]

Pexex Lab

- For more options, use x/nfu [addr] where

– n specifies how much memory to display (in terms of the unit specified by u), default 1

– f specifies the format (ie decimal, hexadecimal, etc.), default hex

– u specifies the unit size for each address (words, bytes, etc.), default word

– Check out the following link for a more precise listing of the parameters:

https://sourceware.org/gdb/onlinedocs/gdb/Memory.html

Pexex Lab

- Also, print out memory based on an address stored in an register:
- Ex: x/20xw $rsp

– Print out 20 words(w) in hex(x) starting from the address stored in %rsp.

Pexex Lab: What needs to be Done

- Set a breakpoint at Ftimes.

- For each instruction until Ftimes completes, examine what each instruction is doing, checking the status of the registers and memory as necessary.

- Answer the questions.

Pexex Lab

For part 2 (examining integer overflow handling), there are three ways:

- gcc -S -fno-asynchronous-unwind-tables <ADDITIONAL FLAGS> testovf.c

– This produces testovf.s which you can read with any text editor.

- gcc -c <ADDITIONAL FLAGS> testovf.c

– This produces testovf.o, which you will have to read using “objdump -d testovf.o”. To save the output of object dump, use “objdump -d testovf.o > testovf.txt”

Pexex Lab

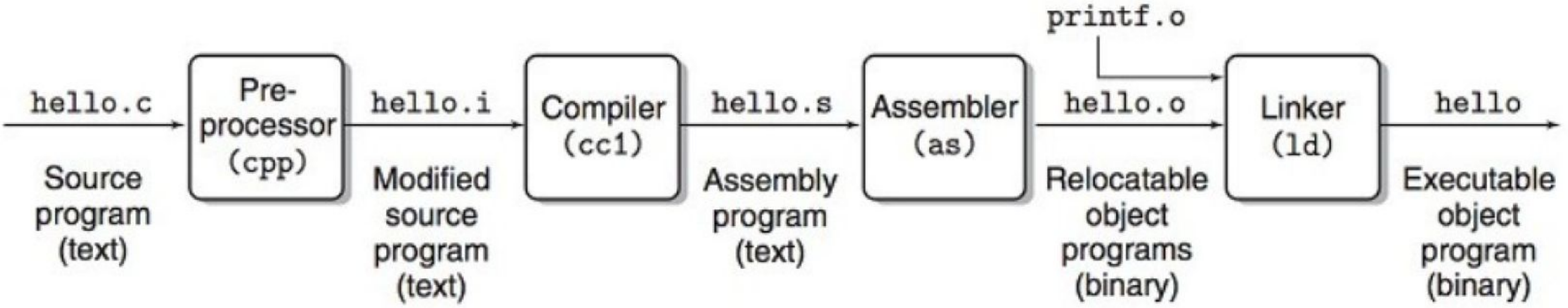
- gcc <ADDITIONAL FLAGS> testovf.c

– This produces a.out, which you can examine with “gdb a.out”.

If you use the -S or -c options, you don't need to include a “main” function. However, if you compile to completion (no -S or -c), you will need to include a main function.

Pexex Lab

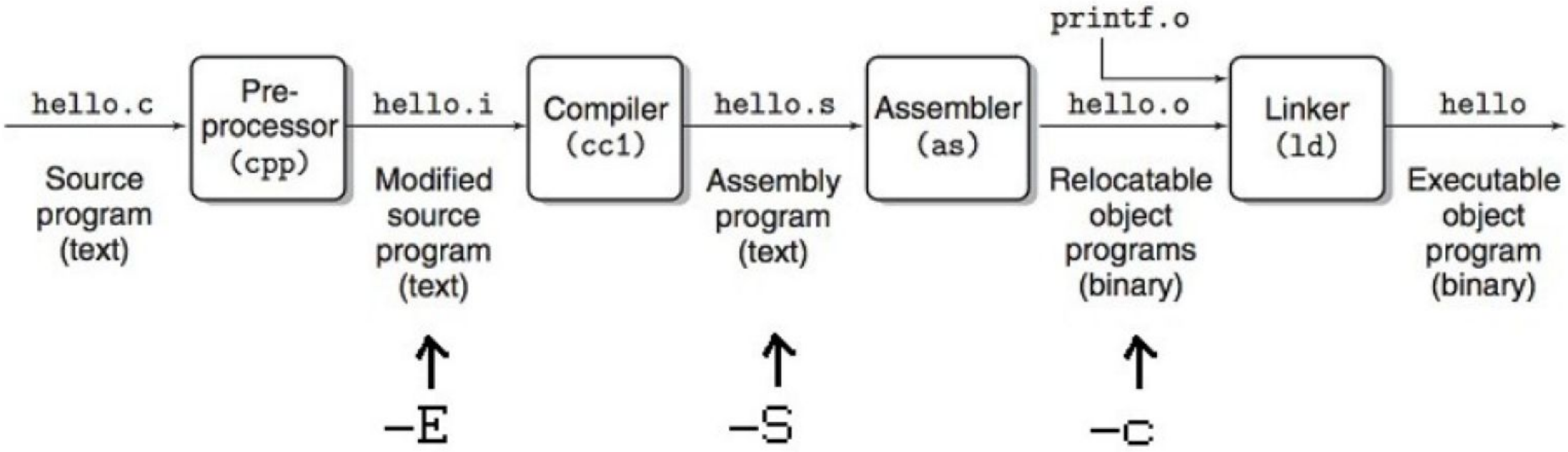
compilation occurs



Pexex Lab

- The result of the Pre-processor step is a modified source with the preprocessor directives(#define, #include) replaced.
- The result of the Compiler step is compiled code, which is readable assembly
- The result of the Assembler step is the assembled code which is a binary file.
- Finally, the result of the linker is a fully executable file.
- gcc allows you to compile up to certain steps

Pexex Lab



- Ex: gcc -E [filename] will get you the modified source file
- Note: Using -E and -S will get you files that you can read with a text editor. To read the output of -c, use objdump.
- To read/disassemble the final executable, use gdb

Problem 3.69 (3rd ed.)

```
typedef struct {
        int first;
        a_struct a[CNT];
        int last;
} b_struct;

void test(long i, b_struct *bp){
        int n = bp->first + bp->last;
        a_struct *ap = &bp->a[i];
        ap->x[ap->idx] = n;
}
```

```
<test>:
mov    0x120(%rsi), %ecx
add    (%rsi), %ecx
lea    (%rdi,%rdi,4), %rax
lea    (%rsi,%rax,8), %rax
mov    0x8(%rax), %rdx
movslq %ecx, %rcx
mov    %rcx, 0x10(%rax,%rdx,8)
ret
```

1. What is CNT?
2. What is the declaration of a_struct? (Assume that the only fields in this structure are idx and x, both containing signed values)

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov    0x120(%rsi), %ecx
[2] add    (%rsi), %ecx
[3] lea    (%rdi,%rdi,4), %rax
[4] lea    (%rsi,%rax,8), %rax
[5] mov    0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

- %rsi is b_struct *bp
- We use bp for “int n = bp->first + bp->last;”
- [1+2] perform *(bp+0x120) + *(bp)

That looks like a match

- So bp->last is at bp + 288
- How much space does bp->a take up?

Problem 3.69 (3rd ed.)

- How much space does bp->a take up?
    - 284 bytes (the 4 bytes of bp->first (int) + the 284 bytes) of bp->a make an offset of 284).

This means sizeof(a_struct) * CNT = 284.

Right?

Problem 3.69 (3rd ed.)

- How much space does bp->a take up?
    - 284 bytes (the 4 bytes of bp->first (int) + the 284 bytes) of bp->a make an offset of 284).

This means sizeof(a_struct) * CNT = 284.
Right?

- **Not necessarily.** Due to alignment, bp->last must be 4-aligned. We know that bp->a requires 284 bytes but bp->a could actually be anywhere between 281 to 284 bytes and still have bp->last be at offset 288.
- Who knows, maybe there's padding between bp->first and bp-> a. If this were true, what would this say about the data types within an instance of a_struct?

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov 0x120(%rsi), %ecx
[2] add (%rsi), %ecx
[3] lea (%rdi, %rdi, 4), %rax
[4] lea (%rsi, %rax, 8) %rax
[5] mov 0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

[3] %rax = i + 4*i = 5i
[4] %rax = bp + 40*i
[5] %rdx = *(bp + 40*i + 8)

What's going on here?

- The ‘i’ is the index used to access &bp->a[i]. That’s probably what this expression is doing.
- If we use i to access elements of the a_struct, what does this say about the size of a_struct?

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov 0x120(%rsi), %ecx
[2] add (%rsi), %ecx
[3] lea (%rdi, %rdi, 4), %rax
[4] lea (%rsi, %rax, 8) %rax
[5] mov 0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

[3] %rax = i + 4*i = 5i
[4] %rax = bp + 40*i
[5] %rdx = *(bp + 40*i + 8)

What's going on here?

- The ‘i’ is the index used to access &bp->a[i]. That’s probably what this expression is doing.
- If we use i to access elements of the a_struct, what does this say about the size of a_struct?
    - sizeof(a_struct) = 40

  What is the 8 in the expression?

Problem 3.69 (3_{rd} ed.)

[5] %rdx = *(bp + 8 + 40*i)

- What is 8 in the expression?

In order to get to bp->a, we must offset by 8 (even though bp->first is an int). Thus, a_struct has a long in it that forces it to be 8 aligned.

Thus, bp + 8 + 40*i is &bp->a[i]. Thus we can rewrite this.

Problem 3.69 (3rd ed.)

[5] %rdx = *(&bp->a[i])

- We're dereferencing a pointer to a struct though. What significance does that have?

    %rdx is set to the first element in bp->a[i].

- What is the size of the very first element?

Problem 3.69 (3rd ed.)

[5] %rdx = *(&bp->a[i])

- We're dereferencing a pointer to a struct though. What significance does that have?

    %rdx is set to the first element in bp->a[i].

- What is the size of the very first element?

    It should be 8 bytes.

- Since &bp->a[i] is assigned to ap:

    [5] %rdx = *(ap);

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov    0x120(%rsi), %ecx
[2] add    (%rsi), %ecx
[3] lea    (%rdi, %rdi, 4), %rax
[4] lea    (%rsi, %rax, 8) %rax
[5] mov    0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

[5] %rdx = *(ap)

[6] %rcx = n

[7] *(16 + bp + 40i + *(ap)*8) = n

%rax %rdx

Problem 3.69 (3rd ed.)

[7] *(16 + bp + 40i + *(ap)*8) = n
=> *(bp + 8 + 40i + 8 + (*ap) * 8) = n
=> *(ap+8+(*ap)*8)=n

- This line matches up with: **ap->x[ap->idx] = n;**

- Thus, *ap is ap->idx. This implies the first element is the ap->idx.

- If this were true, then in order to access ap->x, we would need to offset ap by an additional 8 bytes to skip over ap->idx.

- Oh look: *(ap + ***8*** + (*ap)*8) = n

Problem 3.69 (3rd ed.)

*(ap + 8 + (*ap)*8) = n

```
struct a_struct {
        long idx;
        <?> x[?];
}
```

What is the data type of array a_struct->x?

Problem 3.69 (3rd ed.)

*(ap + 8 + (*ap)*8) = n

```
struct a_struct {
	long idx;
	<?> x[?];
}
```

What is the data type of array a_struct->x?
It's 8 bytes. *ap is idx and in order to use it to index into x, we multiply idx by 8. Thus:

```
struct a_struct {
long idx;
long x[?];
}
```

Problem 3.69 (3rd ed.)

```
struct a_struct {
        long idx;
        <?> x[?];
}
```

What is the length of x?

Recall sizeof(a_struct) is 40 bytes. Thus:

```
struct a_struct {
        long idx;
        long x[4];
}
```

Problem 3.69 (3rd ed.)

```
struct a_struct {
	long idx;
	<?> x[?];
}
```

Let's do a final consistency check and determine CNT. Recall that in b_struct:

Offset 0: int first
Offset ?: a_struct a[CNT];
Offset 288: int last

If this a_struct is correct, the offset would need to be 8. Thus, the array of a would take 280.

CNT = 280 bytes / 40 bytes per a_struct = 7

Floating Point

- So far, have worked with integer data types
  - signed: two's complement
  - unsigned
- Integers: 0, 1, 42, -101, 9001

How to represent a non-integer, like 0.5?

Answer: Floating Point Representation!

Floating Point (IEEE 754)

- Goal: Represent **rational** numbers with a **fixed** number of bits
  - float: 32 bits "single" precision
  - double: 64 bits "double" precision

Floating Point:

Sign bit	"Exponent" Field	"Fraction" Field
1 bit	8 bits	23 bits

Two Types of FP

- There are two different "types" of a floating point number
- Case 1: "Normalized"
    - Common case. Represent large and moderately-small values.
- Case 2: "Denormalized"
    - Represent very small values (close to 0).

Case 1: Normalized

1. Normalized

$$V = (-1)^s \times M \times 2^E$$

s = sign bit
M = 1 + f
E = e - Bias

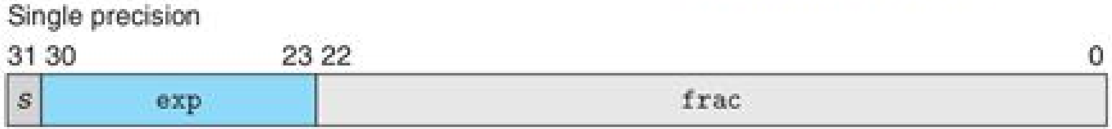
Bias = $2^{k-1} - 1$ = $\begin{cases} \text{127 for single} \\ \text{1023 for double} \end{cases}$

k is # bits in exp

Case 1: Normalized

Bias = $2^{k-1} - 1 = \begin{cases} 127 \text{ for single} \\ 1023 \text{ for double} \end{cases}$

k is # bits in exp



`0100 0010 0010 1000 0000 0000 0000 0000`

What is V?

exp = 100 0010 0 = 0x84 = 8*16 + 4 = 132

f = 010 1000 0000 0000 0000 0000

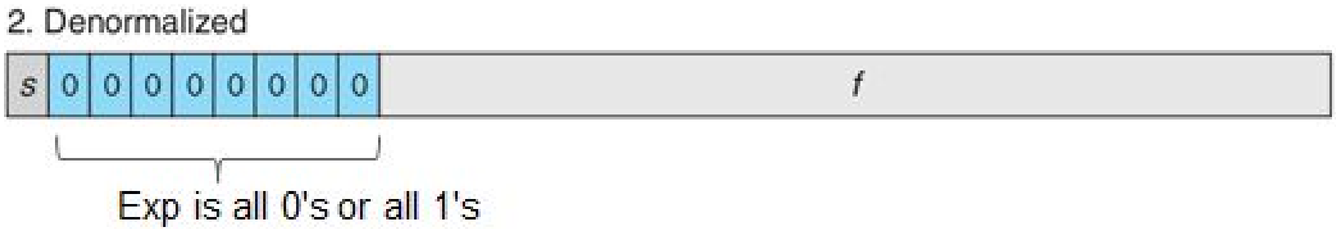
= 0*2^(-1) + 1*2^(-2) + 0*2^(-3) + 1*2^(-4) = 0.3125

V = (-1)^0 * (1 + 0.3125) * 2^(132 - 127) = **42.0**

s = sign bit
M = 1 + f
E = e - Bias

$$V = (-1)^s \times M \times 2^E$$

Case 2: Denormalized



$$V = (-1)^s \times M \times 2^E$$

s = sign bit
M = f
E = 1 - Bias

Bias = $2^{k-1} - 1$ = $\begin{cases} \text{127 for single} \\ \text{1023 for double} \end{cases}$

k is # bits in exp

Case 2: Denormalized

Bias $2^{k-1}-1$ = 127 for single, 1023 for double

k is # bits in exp

1000 0000 0010 1100 0000 0000 0000 0000

exp = 000 0000 0 = 0
f = 010 1100 0000 0000
0000 0000

What is V?

= 1*2^(-2) + 1*2^(-4) + 1*2^(-5) = 0.34375
V = (-1)^1 * (0.34375) * 2^(1-127)
= -4.0407618309516 13e-39

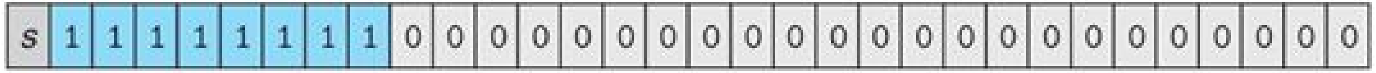
s = sign bit
M = f
E = 1 - Bias

$$V = (-1)^s \times M \times 2^E$$

Case 3: Special Values

- Represent infinity, NaN via certain bit patterns

3a. Infinity



3b. NaN

Note: +Inf and -Inf are different (sign bit).

Case 3: Special Values

- Can represent 0.0 in two ways!
    - All bits 0, and sign bit is 1: -0.0
    - All bits 0, and sign bit is 0: +0.0

Example: Largest/Smallest

- Suppose we use an 8-bit floating-point format. There are 4 exponent bits, and 3 fraction bits.

What is the bias?

`2^(4-1)-1 = 7`

What is the smallest/largest positive value?

```
Smallest: 0 0000 001
Largest: 0 1110 111
```

How to represent 1.0?

`0 0111 000`

Rounding

- Floating point still can't represent every rational number exactly
  - Why? Finite number of bits (32, 64)
- IEEE standard defines several **rounding modes**

Four Rounding Modes

Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$−1.50
Round-to-even	\$1	\$2	\$2	\$2	\$−2
Round-toward-zero	\$1	\$1	\$1	\$2	\$−1
Round-down	\$1	\$1	\$1	\$2	\$−2
Round-up	\$2	\$2	\$2	\$3	\$−1

Figure 2.36 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

FP Operations

- After every operation on two floating point values, a round is performed.
  - Ex: (f+g) => round(f+g)

FP: Addition

- Addition commutes correctly
  - `(f+g) = (g+f)`
- Addition is generally *not* associative
  - `(3.14 + 1e10)-1e10 = 0.0`
  - `3.14 + (1e10-1e10) = 3.14`

FP: Multiplication

- Generally not associative
  - `(1e20*1e20)*1e-20 = +Inf`
  - `1e20*(1e20*1e-20) = 1e20`
- Does not distribute over addition

Exercise

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0:

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

Exercise

We assume that the value `1e400` overflows to infinity.

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

C FP: Casting Rules

- In C, exists float and double data types
  - Can cast to/from float types to integer types.
  - Can lose information due to rounding/truncation!

```
#include <stdio.h>
int main() {
    float v = 42.675;
    int foo = (int) v;
    printf("foo is: %d\n", foo);
    return 0;
}
```

```
$ gcc -o code code.c
$ ./code
foo is: 42
```

Casting Rules Quiz

	Exact conversion?	Can overflow/underflow?
int -> float		
int -> double		
float -> double		
double -> float		
double -> int		
float -> int		

Casting Rules Quiz

	Exact conversion?	Can overflow?
int -> float	No! Float can't repr very large ints.	No
int -> double	Yes	No
float -> double	Yes	No
double -> float	No	Yes
double -> int	No: rounded toward zero (1.99 -> 1, -1.99 -> -1)	Yes
float -> int	No: rounded toward zero	Yes

References

- Slides modified from DJ Kim, UT Wang and Shikhar Malhotra
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Thank You