Computer Systems Fundamentals

floating_types.c

Print size and min and max values of floating point types

```
#include <stdio.h>
#include <float.h>

int main(void) {

    float f;
    printf("float       %2lu bytes  min=%-12g  max=%g\n", sizeof f, FLT_MIN, FLT_MAX);
    double d;
    printf("double      %2lu bytes  min=%-12g  max=%g\n", sizeof d, DBL_MIN, DBL_MAX);
    long double l;
    printf("long double %2lu bytes  min=%-12Lg  max=%Lg\n", sizeof l, LDBL_MIN, LDBL_MAX);

    return 0;
}
```

double_imprecision.c

The value 0.1 can not be precisely represented as a double
As a result b != 0

```
#include <stdio.h>

int main(void) {
    double a, b;

    a = 0.1;
    b = 1 - (a + a + a + a + a + a + a + a + a + a);

    if (b != 0) {
        printf("1 != 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1\n");
    }

    printf("b = %g\n", b); // prints 1.11022e-16

    return 0;
}
```

double_catastrophe.c

Demonstrate approximate representation of reals producing error. sometimes if we subtract two approximations which are very close together we can can get a large relative error
correct answer if x == 0.000000011 (1 - cos(x)) / (x * x) is very close to 0.5 code prints 0.917540 which is wrong by a factor of almost two

```
#include <stdio.h>
#include <math.h>

int main(void) {

    double x = 0.000000011;

    double y = (1 - cos(x)) / (x * x);

    printf("correct answer = ~0.5 but y = %lf\n", y);

    return 0;
}
```

double_disaster.c

9007199254740993 is smallest integer which can not be represented exactly as a double

9007199254740993 is smallest integer which can not be represented exactly as a double

The closest double to 9007199254740993 is 9007199254740992.0
As a result loop never terminates
9007199254740992 is 2 to the power of 53
It can not be represented by a int32_t,
It can be represented by int64_t

```
#include <stdio.h>

int main(void) {

    double d = 9007199254740992;

    while (d < 9007199254740999) {
        printf("%lf\n", d); // prints 9007199254740992.000000
        d = d + 1;
    }

    return 0;
}
```

double_not_always.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[]) {
    assert(argc == 2);

    double d = strtod(argv[1], NULL);

    if (d == d) {
        printf("This should be always executed\n");
    } else {
        // will be executed if d is a NaN
        printf("This should never executed\n");
    }

    printf("d=%g\n", d);

    return 0;
}
```

explain_floating_point_representation.c

Print the underlying representation of a float
The float can be supplied as a decimal or a bit-string

```
$ dcc explain_floating_point_representation.c -o explain_floating_point_representation
$ ./explain_floating_point_representation 0.15625

0.15625 is represented as a float (IEEE-754 single-precision) by these bits:

00111110001000000000000000000000

sign | exponent | fraction
   0 | 01111100 | 01000000000000000000000

sign bit = 0
sign = +

raw exponent    = 01111100 binary
                = 124 decimal
actual exponent = 124 - exponent_bias
                = 124 - 127
                = -3

number = +1.01000000000000000000000 binary * 2**-3
       = 1.25 decimal * 2**-3
       = 1.25 * 0.125
       = 0.15625

$ ./explain_floating_point_representation 150.75

150.75 is represented as a float (IEEE-754 single-precision) by these bits:

01000011000101101100000000000000

sign | exponent | fraction
   0 | 10000110 | 00101101100000000000000

sign bit = 0
sign = +

raw exponent    = 10000110 binary
                = 134 decimal
actual exponent = 134 - exponent_bias
                = 134 - 127
                = 7

number = +1.00101101100000000000000 binary * 2**7
       = 1.17773 decimal * 2**7
       = 1.17773 * 128
       = 150.75

$ ./explain_floating_point_representation -96.125

-96.125 is represented as a float (IEEE-754 single-precision) by these bits:

11000010110000000100000000000000

sign | exponent | fraction
```

```
   1 | 10000101 | 10000000100000000000000

sign bit = 1
sign = -

raw exponent    = 10000101 binary
                = 133 decimal
actual exponent = 133 - exponent_bias
                = 133 - 127
                = 6

number = -1.10000000100000000000000 binary * 2**6
       = -1.50195 decimal * 2**6
       = -1.50195 * 64
       = -96.125

$ ./explain_floating_point_representation inf

inf is represented as a float (IEEE-754 single-precision) by these bits:

01111111100000000000000000000000

sign | exponent | fraction
   0 | 11111111 | 00000000000000000000000

sign bit = 0
sign = +

raw exponent    = 11111111 binary
                = 255 decimal
number = +inf

$ ./explain_floating_point_representation 00111101110011001100110011001101
sign bit = 0
sign = +

raw exponent    = 01111011 binary
                = 123 decimal
actual exponent = 123 - exponent_bias
                = 123 - 127
                = -4

number = +1.10011001100110011001101 binary * 2**-4
       = 1.6 decimal * 2**-4
       = 1.6 * 0.0625
       = 0.1

$ ./explain_floating_point_representation 01111111110000000000000000000000
sign bit = 0
sign = +

raw exponent    = 11111111 binary
                = 255 decimal
number = NaN
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <float.h>
#include <string.h>

void display_float(char *argument);
uint32_t get_float_bits(float f);
void print_float_bits(uint32_t bits);
void print_bit_range(uint32_t value, int high, int low);
void print_float_details(uint32_t bits);
uint32_t extract_bit_range(uint32_t value, int high, int low);
uint32_t convert_bitstring_to_uint32(char *bit_string);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        display_float(argv[arg]);
    }
    return 0;
}

// Define the constants used in representation of a float in IEEE 754 single-precision
// https://en.wikipedia.org/wiki/Single-precision_floating-point_format
// explains format

#define N_BITS             32
#define SIGN_BIT           31
#define EXPONENT_HIGH_BIT  30
#define EXPONENT_LOW_BIT   23
#define FRACTION_HIGH_BIT  22
#define FRACTION_LOW_BIT    0

#define EXPONENT_OFFSET   127
#define EXPONENT_INF_NAN  255

void display_float(char *argument) {
    uint32_t bits;

    // is this argument a bit string or a float?
    if (strlen(argument) > N_BITS - 4 && strspn(argument, "01") == N_BITS) {
        bits = convert_bitstring_to_uint32(argument);
    } else {
        float number = strtof(argument, NULL);
        bits = get_float_bits(number);
        printf("\n%s is represented as a float (IEEE-754 single-precision) by these bits:\n\n", argument);
        print_float_bits(bits);
    }

    print_float_details(bits);
}

void print_float_details(uint32_t bits) {
    uint32_t sign_bit = extract_bit_range(bits, SIGN_BIT, SIGN_BIT);
    uint32_t fraction_bits = extract_bit_range(bits, FRACTION_HIGH_BIT, FRACTION_LOW_BIT);
    uint32_t exponent_bits = extract_bit_range(bits, EXPONENT_HIGH_BIT, EXPONENT_LOW_BIT);

    int sign_char, sign_value;

    if (sign_bit == 1) {
        sign_char = '-';
        sign_value = -1;
    } else {
        sign_char = '+';
        sign_value = 1;
    }

    int exponent = exponent_bits - EXPONENT_OFFSET;

    printf("sign bit = %d\n", sign_bit);
    printf("sign = %c\n\n", sign_char);
```

```
    printf("raw exponent    = ");
    print_bit_range(bits, EXPONENT_HIGH_BIT, EXPONENT_LOW_BIT);
    printf(" binary\n");
    printf("                = %d decimal\n", exponent_bits);

    int implicit_bit = 1;

    // handle special cases of +infinity, -infinity
    // and Not a Number (NaN)
    if (exponent_bits == EXPONENT_INF_NAN) {
        if (fraction_bits == 0) {
            printf("number = %cinf\n\n", sign_char);
        } else {
            // https://en.wikipedia.org/wiki/NaN
            printf("number = NaN\n\n");
        }
        return;
    }

    if (exponent_bits == 0) {
        // if the exponent bits are zero its a special case
        // called a denormal number
        // https://en.wikipedia.org/wiki/Denormal_number
        implicit_bit = 0;
        exponent++;
    }

    printf("actual exponent = %d - exponent_bias\n", exponent_bits);
    printf("                = %d - %d\n", exponent_bits, EXPONENT_OFFSET);
    printf("                = %d\n\n", exponent);

    printf("number = %c%d.", sign_char, implicit_bit);
    print_bit_range(bits, FRACTION_HIGH_BIT, FRACTION_LOW_BIT);
    printf(" binary * 2**%d\n", exponent);

    int fraction_size = FRACTION_HIGH_BIT - FRACTION_LOW_BIT + 1;
    double fraction_max = ((uint32_t)1) << fraction_size;
    double fraction = implicit_bit + fraction_bits / fraction_max;

    fraction *= sign_value;

    printf("       = %g decimal * 2**%d\n", fraction,  exponent);
    printf("       = %g * %g\n", fraction, exp2(exponent));
    printf("       = %g\n\n", fraction * exp2(exponent));
}

union overlay_float {
    float f;
    uint32_t u;
};

// return the raw bits of a float
uint32_t get_float_bits(float f) {
    union overlay_float overlay;
    overlay.f = f;
    return overlay.u;
}

// print out the bits of a float
void print_float_bits(uint32_t bits) {
    print_bit_range(bits, 8 * sizeof bits - 1, 0);
    printf("\n\n");
    printf("sign | exponent | fraction\n");
    printf("   ");
    print_bit_range(bits, SIGN_BIT, SIGN_BIT);
    printf(" | ");
    print_bit_range(bits, EXPONENT_HIGH_BIT, EXPONENT_LOW_BIT);
    printf(" | ");
    print_bit_range(bits, FRACTION_HIGH_BIT, FRACTION_LOW_BIT);
    printf("\n\n");
}
```

```c
// print the binary representation of a value
void print_bit_range(uint32_t value, int high, int low) {
    for (int i = high; i >= low; i--) {
        int bit = extract_bit_range(value, i, i);
        printf("%d", bit);
    }
}

// extract a range of bits from a value
uint32_t extract_bit_range(uint32_t value, int high, int low) {
    uint32_t mask = (((uint32_t)1) << (high - low + 1)) - 1;
    return (value >> low) & mask;
}

// given a string of 1s and 0s return the correspong uint32_t
uint32_t convert_bitstring_to_uint32(char *bit_string) {
    uint32_t bits = 0;
    for (int i = 0; i < N_BITS && bit_string[i] != '\0'; i++) {
        int ascii_char = bit_string[N_BITS - 1 - i];
        uint32_t bit = ascii_char != '0';
        bits = bits | (bit << i);
    }
    return bits;
}
```

COMP1521 20T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au
CRICOS Provider 00098G