Chapter 2 Lab Practices: Integer Representations, Arithmetic, Floating-Point, and Additional Bit Manipulation

DS2040, Satyajit Das

January 16, 2025

Introduction

This lab collection covers hands-on programming tasks related to:

- Integer representations (two's complement, overflow, sign extension)
- Integer arithmetic (truncation, division, multiplication)
- Floating-point representation and arithmetic (precision, rounding, Inf, NaN)
- Bit manipulation (shifts, masking, rotating bits, setting/clearing bits)

By completing these programs, students will gain a deeper understanding of how data is represented and manipulated at the machine level.

1 Programs

1. Printing Integer Representations

Purpose: Demonstrate how integers can be displayed in different bases (decimal, octal, hexadecimal).

Listing 1: print_int_representations.c

```c
#include <stdio.h>

int main(void) {
    int num = 100;

    printf("Decimal: %d\n", num);
    printf("Octal: %o\n", num);
    printf("Hexadecimal: %x\n", num);

    return 0;
}
```

Key Concepts:

- %d for decimal
- %o for octal
- %x for hexadecimal

2. Two's Complement Observation

Purpose: Observe how two's complement wraps around when dealing with signed integers.

Listing 2: twos_complement_wrap.c

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
    int x = INT_MAX;
    printf("INT_MAX = %d\n", x);
    x = x + 1; // Overflow on a 2's complement system
    printf("After adding 1 to INT_MAX: %d\n", x);

    int y = INT_MIN;
    printf("INT_MIN = %d\n", y);
    y = y - 1; // Underflow
    printf("After subtracting 1 from INT_MIN: %d\n", y);

    return 0;
}
```

Key Concepts:

- Two's complement overflow/underflow behavior
- INT_MAX and INT_MIN

3. Sign Extension in Arithmetic

Purpose: Demonstrate what happens when a smaller signed integer is promoted to a larger type.

Listing 3: sign_extension.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int16_t smallNum = -12345; // 16-bit signed
    int32_t largeNum = smallNum; // Implicit sign extension

    printf("16-bit smallNum = %d\n", smallNum);
    printf("32-bit largeNum (extended) = %d\n", largeNum);

    return 0;
}
```

Key Concepts:

- Sign extension from smaller to larger type
- Preserving negative values

4. Bitwise Operations on Integers

Purpose: Review `AND`, `OR`, `XOR`, bit-shifts, and see how they affect integer values.

Listing 4: integer_bitwise_ops.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint32_t a = 0xF0F0F0F0;
    uint32_t b = 0xAAAA5555;

    printf("a = 0x%08X\n", a);
    printf("b = 0x%08X\n", b);

    printf("a & b = 0x%08X\n", (a & b));
    printf("a | b = 0x%08X\n", (a | b));
    printf("a ^ b = 0x%08X\n", (a ^ b));
    printf("a << 4 = 0x%08X\n", (a << 4));
    printf("b >> 4 = 0x%08X\n", (b >> 4));

    return 0;
}
```

Key Concepts:

- Bitwise &, |, ^
- Left and right shifting («, »)
- Hexadecimal display format (%08X)

5. Endianness Check

Purpose: Illustrates how to check if a system is little-endian or big-endian using a small integer.

Listing 5: endianness_check.c

```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0x12345678;
    unsigned char *ptr = (unsigned char *)&x;

    printf("Memory representation: ");
    for(int i = 0; i < 4; i++) {
        printf("%02X ", ptr[i]);
```

```c
    }
    printf("\n");

    if (ptr[0] == 0x78) {
        printf("System is Little-Endian.\n");
    } else {
        printf("System is Big-Endian.\n");
    }

    return 0;
}
```

Key Concepts:

- Interpreting memory byte by byte
- Detecting endianness by examining the least significant byte

6. Unsigned Arithmetic Overflow

Purpose: Show what happens when you exceed the maximum of an unsigned type.

Listing 6: unsigned_overflow.c

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
    unsigned int max = UINT_MAX;
    printf("UINT_MAX = %u\n", max);

    unsigned int overflowed = max + 1;
    printf("After adding 1 to UINT_MAX: %u\n", overflowed);

    return 0;
}
```

Key Concepts:

- Unsigned integer range
- Wrap-around behavior (mod 2^n)

7. Multiplication and Division Truncation

Purpose: Demonstrate how integer multiplication and division behave with truncation and overflow.

Listing 7: int_mult_div.c

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
    int a = 20000;
```

```c
    int b = 20000;
    long long result = (long long)a * b; // to see exact product

    printf("a * b (int) = %d (overflow expected)\n", a * b);
    printf("a * b (long long) = %lld\n", result);

    int numerator = 7;
    int denominator = 2;
    printf("Integer division of 7 / 2 = %d\n", numerator / denominator);

    return 0;
}
```

Key Concepts:

- Overflow when `int` is too small
- Truncation in integer division

8. Fixed-Width Integer Arithmetic

Purpose: Explore arithmetic with types like `int8_t`, `int16_t`, `int32_t`, which can overflow more easily.

Listing 8: fixed_width_integers.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int8_t x8  = 120;  // near INT8_MAX is 127
    int16_t x16 = 30000; // near INT16_MAX is 32767

    printf("x8 = %d\n", x8);
    x8 = x8 + 10; // Overflow might occur
    printf("After adding 10: x8 = %d\n", x8);

    printf("x16 = %d\n", x16);
    x16 = x16 + 5000; // Might overflow
    printf("After adding 5000: x16 = %d\n", x16);

    return 0;
}
```

Key Concepts:

- Fixed-width integers from `<stdint.h>`
- Overflow in smaller data types

9. Simple Floating-Point Representation

Purpose: Show how floats store approximate values and compare differences with doubles.

Listing 9: float_rep.c

```c
#include <stdio.h>

int main(void) {
    float f = 3.141592653589793f; // single precision
    double d = 3.141592653589793; // double precision

    printf("float f: %.9f\n", f);
    printf("double d: %.15f\n", d);

    return 0;
}
```

Key Concepts:

- Single precision vs. double precision
- Displaying more decimal places than the type can store accurately

10. Rounding Errors in Floating-Point Arithmetic

Purpose: Demonstrate how floating-point addition/subtraction leads to rounding errors.

Listing 10: float_rounding.c

```c
#include <stdio.h>

int main(void) {
    float sum = 0.0f;
    for(int i = 0; i < 1000; i++) {
        sum += 0.01f;
    }
    printf("Expected 10.0, got %f\n", sum);

    return 0;
}
```

Key Concepts:

- Floating-point rounding errors
- Accumulated precision loss

11. Casting Between Floating-Point and Integer

Purpose: Illustrate truncation and rounding when casting between `float`/`double` and integers.

Listing 11: float_int_cast.c

```c
#include <stdio.h>

int main(void) {
    float f = 3.99f;
    int i = (int)f; // cast to int truncates toward zero
```

```c

    printf("float f = %.2f\n", f);
    printf("After casting to int: i = %d\n", i);

    double d = 123456789.0;
    int j = (int)d; // might exceed int range if too large
    printf("double d = %.0f\n", d);
    printf("After casting to int: j = %d\n", j);

    return 0;
}
```

Key Concepts:

- Truncation from float/double to int
- Potential overflow if float/double is too large

12. Exploring `printf` Format Specifiers for Floating-Point

Purpose: Practice printing floating-point numbers in various formats.

Listing 12: float_format_specifiers.c

```c
#include <stdio.h>

int main(void) {
    double val = 12345.6789;

    printf("Default: %f\n", val);
    printf("Fixed decimal (2 digits): %.2f\n", val);
    printf("Scientific notation: %e\n", val);
    printf("Hexadecimal notation: %a\n", val);

    return 0;
}
```

Key Concepts:

- `%f`, `%e`, `%a` format specifiers
- Scientific and hexadecimal floating-point representation

13. Floating-Point Comparisons Pitfall

Purpose: Demonstrate why direct equality checks on floating-point numbers are problematic.

Listing 13: float_comparison.c

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

int main(void) {
    double x = 0.1 + 0.2;
```

```c
    double y = 0.3;

    if (x == y) {
        printf("x == y\n");
    } else {
        printf("x != y (due to floating-point precision)\n");
    }

    // A better way:
    if (fabs(x - y) < 1e-9) {
        printf("x and y are close enough\n");
    } else {
        printf("x and y differ by more than 1e-9\n");
    }

    return 0;
}
```

Key Concepts:

- Precision errors in floating-point arithmetic
- Using an epsilon for comparisons

14. Checking Floating-Point Range and Infinity

Purpose: Understand how floating-point numbers handle very large values or division by zero.

Listing 14: float_infinity.c

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

int main(void) {
    double huge = 1.0e308;
    double bigger = huge * 1000.0; // Likely overflow to infinity

    printf("huge = %e\n", huge);
    printf("bigger = %e\n", bigger);

    double zero = 0.0;
    double inf = 1.0 / zero;
    printf("1.0 / 0.0 = %f\n", inf); // Infinity

    double nanVal = 0.0 / zero;
    printf("0.0 / 0.0 = %f (NaN)\n", nanVal);

    return 0;
}
```

Key Concepts:

- IEEE 754 behavior for overflow (Inf) and invalid operations (NaN)
- Distinguishing finite numbers vs. `inf`, `NaN`

15. Mixing Integer and Floating-Point Arithmetic

Purpose: Examine how C performs type promotion when expressions have both integer and floating types.

Listing 15: mixed_arithmetic.c

```c
#include <stdio.h>

int main(void) {
    int i = 5;
    float f = 2.5f;

    float result1 = i + f;       // i promoted to float
    double result2 = (double)i / 2; // i stays int, but result is double

    printf("i + f = %.2f\n", result1);
    printf("(double)i / 2 = %.2f\n", result2);

    // Another subtlety:
    int j = (int)(i * f); // i*f is float, then truncated
    printf("(int)(i * f) = %d\n", j);

    return 0;
}
```

Key Concepts:

- Integer-to-float promotion rules
- Truncation vs. rounding

16. Reversing Bits in an Unsigned Integer

Purpose: Practice manipulating bits to reverse the bit order of an unsigned integer.

Listing 16: bit_reverse.c

```c
#include <stdio.h>
#include <stdint.h>

uint32_t reverse_bits(uint32_t x) {
    uint32_t reversed = 0;
    for(int i = 0; i < 32; i++) {
        reversed <<= 1;
        reversed |= (x & 1);
        x >>= 1;
    }
    return reversed;
}

int main(void) {
    uint32_t num = 0x12345678;
    uint32_t rev = reverse_bits(num);

    printf("Original: 0x%08X\n", num);
    printf("Reversed: 0x%08X\n", rev);

    return 0;
}
```

Key Concepts:

- Looping to shift and accumulate bits
- Bitwise masking (`& 1`)

17. Rotating Bits Left and Right

Purpose: Demonstrate how to rotate bits around in a 32-bit integer (different from shifting).

Listing 17: bit_rotate.c

```c
#include <stdio.h>
#include <stdint.h>

uint32_t rotate_left(uint32_t x, int n) {
    return (x << n) | (x >> (32 - n));
}

uint32_t rotate_right(uint32_t x, int n) {
    return (x >> n) | (x << (32 - n));
}

int main(void) {
    uint32_t val = 0xABCDEF12;
```

```c

    printf("Original: 0x%08X\n", val);
    printf("Rotated L4: 0x%08X\n", rotate_left(val, 4));
    printf("Rotated R4: 0x%08X\n", rotate_right(val, 4));

    return 0;
}
```

Key Concepts:

- Difference between rotate and shift (shifts introduce zeros)
- `|` to combine shifted portions

18. Counting Set Bits in an Integer

Purpose: Show how to count the number of 1-bits in a given integer (a.k.a. the Hamming weight).

Listing 18: count_set_bits.c

```c
#include <stdio.h>
#include <stdint.h>

int count_set_bits(uint32_t x) {
    int count = 0;
    while (x) {
        count += (x & 1);
        x >>= 1;
    }
    return count;
}

int main(void) {
    uint32_t num = 0xF0F0F0F0;
    int bits = count_set_bits(num);

    printf("Number = 0x%08X\n", num);
    printf("Set bits = %d\n", bits);

    return 0;
}
```

Key Concepts:

- Iterative approach to count bits
- Bitwise `& 1` and right shift

19. Checking the Sign with Bits (Signed vs. Unsigned)

Purpose: Use bitwise operations to check if a signed integer is negative and compare with an unsigned interpretation.

Listing 19: check_sign.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int32_t s = -42; // signed
    uint32_t u = (uint32_t)s; // re-interpret bits as unsigned

    printf("Signed s = %d\n", s);
    printf("Unsigned u = %u\n", u);

    // Checking sign bit manually (assuming 32-bit two's complement)
    int sign_bit = (s >> 31) & 1;
    printf("Sign bit of s: %d\n", sign_bit);

    return 0;
}
```

Key Concepts:

- Reinterpreting the same bits as signed vs. unsigned
- Extracting the sign bit with a right shift

20. Clearing and Setting Specific Bits

Purpose: Practice using masks to clear (set to 0) and set (to 1) specific bits within an integer.

Listing 20: bit_set_clear.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint32_t x = 0xFF00FF00;
    printf("Original x = 0x%08X\n", x);

    // Clear the lower 8 bits
    uint32_t mask_clear = 0xFFFFFF00; // 11111111 11111111 11111111 00000000
    x = x & mask_clear;
    printf("After clearing lower 8 bits = 0x%08X\n", x);

    // Set bits 8 to 15
    uint32_t mask_set = 0x0000FF00;
    x = x | mask_set;
    printf("After setting bits 8-15 = 0x%08X\n", x);

    return 0;
}
```

Key Concepts:

- Constructing masks for specific bits

- `&` to clear bits, `|` to set bits

21. Converting Signed to Unsigned and Observing Behavior

Purpose: Show how assigning a negative signed integer to an unsigned variable can change its interpretation.

Listing 21: signed_to_unsigned.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int32_t negativeVal = -12345;
    uint32_t convertedVal = (uint32_t)negativeVal;

    printf("signed int: %d\n", negativeVal);
    printf("unsigned int: %u\n", convertedVal);
    printf("as hex: 0x%08X\n", convertedVal);

    return 0;
}
```

Key Concepts:

- Bitwise re-interpretation
- Large unsigned values representing negative signed integers

22. Extracting Bytes from a 32-bit Integer

Purpose: Show how to extract individual bytes (e.g., for network protocols or endianness checks).

Listing 22: extract_bytes.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint32_t val = 0x12345678;
    unsigned char *ptr = (unsigned char *)&val;

    printf("val = 0x%08X\n", val);
    printf("Byte 0: 0x%02X\n", ptr[0]);
    printf("Byte 1: 0x%02X\n", ptr[1]);
    printf("Byte 2: 0x%02X\n", ptr[2]);
    printf("Byte 3: 0x%02X\n", ptr[3]);

    return 0;
}
```

Key Concepts:

- Using a `char *` pointer to access underlying bytes
- Endianness and byte ordering

23. Shifting a Negative Number and Observing Sign Extension

Purpose: Demonstrate arithmetic shift vs. logical shift on a negative number (in C, right shift on signed is implementation-defined, but usually arithmetic).

Listing 23: negative_shift.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int32_t neg = -1;      // 0xFFFFFFFF in two's complement
    uint32_t uneg = (uint32_t)neg;

    printf("neg (signed) = %d\n", neg);
    printf("uneg (unsigned) = %u\n", uneg);

    // Right shifting a signed negative
    int32_t shiftedNeg = neg >> 1; // typically arithmetic shift
    printf("neg >> 1 = %d (0x%08X)\n", shiftedNeg, (uint32_t)shiftedNeg);

    // Right shifting the unsigned version (logical shift)
    uint32_t shiftedUNeg = uneg >> 1;
    printf("uneg >> 1 = %u (0x%08X)\n", shiftedUNeg, shiftedUNeg);

    return 0;
}
```

Key Concepts:

- Arithmetic vs. logical shifts
- Sign extension on right shift of signed negative numbers
- Implementation-defined aspects in C

Compilation and Practice Tips

- **Compile Each Program:**

```
gcc filename.c -o program
./program
```

- **Experimentation:**
  - Change constants and data types to see different behaviors (e.g., `int16_t` vs. `int32_t`).
  - Print intermediate results or debug info to trace calculations.
- **Further Practice:**
  - Implement functions to convert integers to binary strings.
  - Parse and print IEEE 754 binary components (sign, exponent, mantissa).
  - Test corner cases with large values, small values, negative values, or extreme floating-point numbers.