

DPST1092 23T2 — Bitwise Operators

<https://www.cse.unsw.edu.au/~dp1092/23T2/>

Bitwise Operators

Everything in C really is just 1's and 0's under the hood!

- eg. the number 42 is `0b00101010`
- eg. the ascii character '#' is `0b00100011`
- eg. the floating point 3.14 is `0b01000000010010001111010111000011`

C provides 6 operators which operate on individual bits of values

```
x & y    // bitwise and
x | y    // bitwise or
x ^ y    // bitwise exclusive-or (XOR)
~ x      // bitwise not
x << n   // left shift
x >> n   // right shift
```

Bitwise AND: &

The **&** operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical AND on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

00100111	AND		0	1
& 11100011	----		-----	

00100011			0	0
			1	0
			0	1

Used for e.g. checking whether a bit is set

Checking for Odd Numbers

The obvious way to check for odd numbers in C

```
int is_odd(int n) {  
    return n % 2 != 0;  
}
```

We can use **&** to achieve the same thing:

```
int is_odd(int n) {  
    return n & 1;  
}
```

Bitwise OR: |

The | operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical OR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

00100111	OR		0	1
11100011	----		-----	
-----	0		0	1
11100111	1		1	1

Used for e.g. ensuring that a bit is set

Bitwise NEG: ~

The ~ operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- performs logical negation of each bit
- result contains same number of bits as input

Example:

~ 00100111	NEG 0 1
-----	---- -----
11011000	1 0

Used for e.g. creating useful bit patterns

Bitwise XOR: ^

The ^ operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical XOR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
00100111    XOR | 0  1
^ 11100011    ----|-----
-----      0  | 0  1
11000100     1  | 1  0
```

Used in e.g. generating hashes, graphics operations, cryptography

Exercise: Bit Operations

What are the values of the following:

- $(01010101 \ \& \ 10101010)$
- $(01010101 \ | \ 10101010)$
- $(x \ \& \ \sim x)$
- $(x \ | \ \sim x)$

Assume we have a variable `uint8_t x`. Write a line/s of code to

- set the 3rd least significant bit to 1
- set the 3rd least significant bit to 0

Left Shift: <<

The << operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- also takes a small positive integer x
- moves (shifts) each bit x positions to the left
- left-end bit vanishes; right-end bit replaced by zero
- result contains same number of bits as input

Example:

```
00100111 << 2      00100111 << 8
-----
10011100           00000000
```

Right Shift: >>

The >> operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- also takes a small positive integer x
- moves (shifts) each bit x positions to the right
- right-end bit vanishes; left-end bit replaced by zero(*)
- result contains same number of bits as input

Example:

```
00100111 >> 2      00100111 >> 8
-----
00001001           00000000
```

- shifts involving negative values are not portable (implementation defined)
- common source of bugs in CP1521 and elsewhere
- always use unsigned values/variables to be safe/portable.

Exercise: Shifts

Given the declaration: `uint8_t x = 19`. What is the value of

- `x >> 2`
- `x << 1`

Write code to

- multiply `x` by 4 using bit shift operations.
- divide `x` by 8 using bit shift operations.
- calculate 2 power of `n`, for a given `n`.
- print out a given `uint8_t` in binary.

shift_as_multiply.c: using shift to multiply by 2^n

```
$ gcc shift_as_multiply.c print_bits.c -o shift_as_multiply
$ ./shift_as_multiply 4
2 to the power of 4 is 16
In binary it is: 0000000000000000000000000000000010000
$ ./shift_as_multiply 20
2 to the power of 20 is 1048576
In binary it is: 000000000000100000000000000000000000
$ ./shift_as_multiply 31
2 to the power of 31 is 2147483648
In binary it is: 100000000000000000000000000000000000
$
```

print_bits.c: extracting the n-th bit of a value

```
// print the bottom how_many_bits bits of value
void print_bits(uint64_t value, int how_many_bits) {
    // print bits from most significant to least significant
    for (int i = how_many_bits - 1; i >= 0; i--) {
        int bit = get_nth_bit(value, i);
        printf("%d", bit);
    }
}
```

```
// extract the nth bit from a value
int get_nth_bit(uint64_t value, int n) {
    // shift the bit right n bits
    // this leaves the n-th bit as the least significant bit
    uint64_t shifted_value = value >> n;
    // zero all bits except the the least significant bit
    int bit = shifted_value & 1;
    return bit;
}
```

source code for print_bits.c

shift_bug.c: bugs to avoid

```
// int16_t is a signed type (-32768..32767)  
// below operations are undefined for a signed type  
int16_t i;  
i = -1;  
i = i >> 1; // undefined - shift of a negative value  
printf("%d\n", i);  
i = -1;  
i = i << 1; // undefined - shift of a negative value  
printf("%d\n", i);  
i = 32767;  
i = i << 1; // undefined - left shift produces a negative value  
printf("%d\n", i);  
uint64_t j;  
j = 1 << 33; // undefined - constant 1 is an int  
j = ((uint64_t)1) << 33; // ok  
printf("%lu\n", j);
```

source code for shift_bug.c

set_low_bits.c: using << and - to set low n bits

```
$ gcc set_low_bits.c print_bits.c -o n_ones
$ ./set_low_bits 3
The bottom 3 bits of 7 are ones:
0000000000000000000000000000000000111
$ ./set_low_bits 19
The bottom 19 bits of 524287 are ones:
0000000000000011111111111111111111111
$ ./set_low_bits 29
The bottom 29 bits of 536870911 are ones:
00011111111111111111111111111111111
```

set_low_bits.c: using << and - to set low n bits

```
int n = strtol(argv[1], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof mask;
assert(n >= 0 && n < n_bits);
mask = 1;
mask = mask << n;
mask = mask - 1;
printf("The bottom %d bits of %u are ones:\n", n, mask);
print_bits(mask, n_bits);
printf("\n");
```

source code for set_low_bits.c

set_bit_range.c: using << and - to set a range of bits

```
$ gcc set_bit_range.c print_bits.c -o set_bit_range
$ ./set_bit_range 0 7
Bits 0 to 7 of 255 are ones:
0000000000000000000000000000000011111111
$ ./set_bit_range 8 15
Bits 8 to 15 of 65280 are ones:
0000000000000000000000001111111100000000
$ ./set_bit_range 8 23
Bits 8 to 23 of 16776960 are ones:
00000000111111111111111100000000
$ ./set_bit_range 1 30
Bits 1 to 30 of 2147483646 are ones:
011111111111111111111111111111110
```

set_bit_range.c: using << and - to set a range of bits

```
int low_bit = strtol(argv[1], NULL, 0);
int high_bit = strtol(argv[2], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof mask;

int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
printf("Bits %d to %d of %u are ones:\n", low_bit, high_bit, mask);
print_bits(mask, n_bits);
printf("\n");
```

source code for set_bit_range.c

extract_bit_range.c: extracting a range of bits

```
$ gcc extract_bit_range.c print_bits.c -o extract_bit_range
$ ./extract_bit_range 4 7 42
Value 42 in binary is:
00000000000000000000000000000000101010
Bits 4 to 7 of 42 are:
0010
$ ./extract_bit_range 10 20 123456789
Value 123456789 in binary is:
00000111010110111100110100010101
Bits 10 to 20 of 123456789 are:
11011110011
```

extract_bit_range.c: extracting a range of bits

```
int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
// get a value with the bits outside the range low_bit..high_bit set to zero
uint32_t extracted_bits = value & mask;
// right shift the extracted_bits so low_bit becomes bit 0
extracted_bits = extracted_bits >> low_bit;
printf("Value %u in binary is:\n", value);
print_bits(value, n_bits);
printf("\n");
printf("Bits %d to %d of %u are:\n", low_bit, high_bit, value);
print_bits(extracted_bits, mask_size);
printf("\n");
```

source code for extract_bit_range.c

print_int_in_hex.c: print an integer in hexadecimal

- write C to print an integer in hexadecimal instead of using:

```
printf("%x", n)
```

```
$ gcc print_int_in_hex.c -o print_int_in_hex
```

```
$ ./print_int_in_hex
```

```
Enter a positive int: 42
```

```
42 = 0x0000002A
```

```
$ ./print_int_in_hex
```

```
Enter a positive int: 65535
```

```
65535 = 0x0000FFFF
```

```
$ ./print_int_in_hex
```

```
Enter a positive int: 3735928559
```

```
3735928559 = 0xDEADBEEF
```

```
$
```

source code for print_int_in_hex.c

print_int_in_hex.c: main

```
int main(void) {
    uint32_t a = 0;
    printf("Enter a positive int: ");
    scanf("%u", &a);
    printf("%u = 0x", a);
    print_hex(a);
    printf("\n");
    return 0;
}
```

source code for print_int_in_hex.c

print_int_in_hex.c: print_hex - extracting digit

```
// sizeof returns number of bytes in n's representation
// each byte is 2 hexadecimal digits
int n_hex_digits = 2 * (sizeof n);
// print hex digits from most significant to least significant
for (int which_digit = n_hex_digits - 1; which_digit >= 0; which_digit--) {
    // shift value across so hex digit we want
    // is in bottom 4 bits
    int bit_shift = 4 * which_digit;
    uint32_t shifted_value = n >> bit_shift;
    // mask off (zero) all bits but the bottom 4 bites
    int hex_digit = shifted_value & 0xF;
    // hex digit will be a value 0..15
    // obtain the corresponding ASCII value
    // "0123456789ABCDEF" is a char array
    // containing the appropriate ASCII values (+ a '\0')
    int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
    putchar(hex_digit_ascii);
}
```

int_to_hex_string.c: convert int to a string of hex digits

- Write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function `snprintf` to do this.

```
$ gcc int_to_hex_string.c -o int_to_hex_string
```

```
$ ./int_to_hex_string
```

```
$ ./int_to_hex_string
```

```
Enter a positive int: 42
```

```
42 = 0x0000002A
```

```
$ ./int_to_hex_string
```

```
Enter a positive int: 65535
```

```
65535 = 0x0000FFFF
```

```
$ ./int_to_hex_string
```

```
Enter a positive int: 3735928559
```

```
3735928559 = 0xDEADBEEF
```

```
$
```

[source code for int_to_hex_string.c](#)

int_to_hex_string.c: main

```
int main(void) {
    uint32_t a = 0;
    printf("Enter a positive int: ");
    scanf("%u", &a);
    char *hex_string = int_to_hex_string(a);
    // print the returned string
    printf("%u = 0x%s\n", a, hex_string);
    free(hex_string);
    return 0;
}
```

source code for int_to_hex_string.c

int_to_hex_string.c: convert int to a string of hex digits

```
// sizeof returns number of bytes in n's representation
// each byte is 2 hexadecimal digits
int n_hex_digits = 2 * (sizeof n);
// allocate memory to hold the hex digits + a terminating 0
char *string = malloc(n_hex_digits + 1);
// print hex digits from most significant to least significant
for (int which_digit = 0; which_digit < n_hex_digits; which_digit++) {
    // shift value across so hex digit we want
    // is in bottom 4 bits
    int bit_shift = 4 * which_digit;
    uint32_t shifted_value = n >> bit_shift;
    // mask off (zero) all bits but the bottom 4 bites
    int hex_digit = shifted_value & 0xF;
    // hex digit will be a value 0..15
    // obtain the corresponding ASCII value
    // "0123456789ABCDEF" is a char array
    // containing the appropriate ASCII values
    int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
    int string_position = n_hex_digits - which_digit - 1;
    string[string_position] = hex_digit_ascii;
}
// 0 terminate the array
string[n_hex_digits] = 0;
return string;
```

source code for int_to_hex_string.c

hex_string_to_int.c: convert hex digit string to int

- As an exercise write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function `strtol` to do this.

```
$ gcc hex_string_to_int.c -o hex_string_to_int
$ gcc hex_string_to_int.c -o hex_string_to_int
$ ./hex_string_to_int 2A
2A hexadecimal is 42 base 10
$ ./hex_string_to_int FFFF
FFFF hexadecimal is 65535 base 10
$ ./hex_string_to_int DEADBEEF
DEADBEEF hexadecimal is 3735928559 base 10
$
```

[source code for hex_string_to_int.c](#)

hex_string_to_int.c: main

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <hexadecimal-number>\n", argv[0]);
        return 1;
    }
    char *hex_string = argv[1];
    uint32_t u = hex_string_to_int(hex_string);
    printf("%s hexadecimal is %u base 10\n", hex_string, u);
    return 0;
}
```

source code for hex_string_to_int.c

hex_string_to_int.c: convert array of hex digits to int

```
uint32_t hex_string_to_int(char *hex_string) {
    uint32_t value = 0;
    for (int i = 0; hex_string[i] != 0; i++) {
        int ascii_hex_digit = hex_string[i];
        int digit_as_int = hex_digit_to_int(ascii_hex_digit);
        value = value << 4;
        value = value | digit_as_int;
    }
    return value;
}
```

source code for hex_string_to_int.c

hex_string_to_int.c: convert single hex digit to int

```
int hex_digit_to_int(int ascii_digit) {
    if (ascii_digit >= '0' && ascii_digit <= '9') {
        // the ASCII characters '0' .. '9' are contiguous
        // in other words they have consecutive values
        // so subtract the ASCII value for '0' yields the corresponding integer
        return ascii_digit - '0';
    }
    if (ascii_digit >= 'A' && ascii_digit <= 'F') {
        // for characters 'A' .. 'F' obtain the
        // corresponding integer for a hexadecimal digit
        return 10 + (ascii_digit - 'A');
    }
    fprintf(stderr, "Bad digit '%c'\n", ascii_digit);
    exit(1);
}
```

source code for hex_string_to_int.c

xor.c: fun with xor

```
int xor_value = strtol(argv[1], NULL, 0);
if (xor_value < 0 || xor_value > 255) {
    fprintf(stderr, "Usage: %s <xor-value>\n", argv[0]);
    return 1;
}
int c;
while ((c = getchar()) != EOF) {
    //     exclusive-or
    //     ^  | 0  1
    //     ----|-----
    //     0  | 0  1
    //     1  | 1  0
    int xor_c = c ^ xor_value;
    putchar(xor_c);
}
```

source code for xor.c

xor .c: fun with xor

```
$ echo Hello Andrew|xor 42
bOFFE
kDNXO] $ echo Hello Andrew|xor 42|cat -A
bOFFE$
kDNXO] $
$ echo Hello |xor 42
bOFFE $ echo -n 'bOFFE '|xor 42
Hello
$ echo Hello|xor 123|xor 123
Hello
$
```


pokemon.c: using an int to represent a set of values

```
#define FIRE_TYPE      0x0001
#define FIGHTING_TYPE 0x0002
#define WATER_TYPE     0x0004
#define FLYING_TYPE    0x0008
#define POISON_TYPE    0x0010
#define ELECTRIC_TYPE  0x0020
#define GROUND_TYPE    0x0040
#define PSYCHIC_TYPE   0x0080
#define ROCK_TYPE      0x0100
#define ICE_TYPE       0x0200
#define BUG_TYPE       0x0400
#define DRAGON_TYPE    0x0800
#define GHOST_TYPE     0x1000
#define DARK_TYPE      0x2000
#define STEEL_TYPE     0x4000
#define FAIRY_TYPE     0x8000
```

source code for pokemon.c

pokemon.c: using an int to represent a set of values

- simple example of a single integer specifying a set of values
- interacting with hardware often involves this sort of code

```
uint16_t our_pokemon = BUG_TYPE | POISON_TYPE | FAIRY_TYPE;
```

```
// example code to check if a pokemon is of a type:
```

```
if (our_pokemon & POISON_TYPE) {  
    printf("Poisonous\n"); // prints  
}  
if (our_pokemon & GHOST_TYPE) {  
    printf("Scary\n"); // does not print  
}
```

source code for pokemon.c

pokemon.c: using an int to represent a set of values

```
// example code to add a type to a pokemon  
our_pokemon |= GHOST_TYPE;  
// example code to remove a type from a pokemon  
our_pokemon &= ~ POISON_TYPE;
```

```
printf(" our_pokemon type (2)\n");  
if (our_pokemon & POISON_TYPE) {  
    printf("Poisonous\n"); // does not print  
}  
if (our_pokemon & GHOST_TYPE) {  
    printf("Scary\n"); // prints  
}
```

source code for pokemon.c