Information Processing 2 — Week 5 exercises

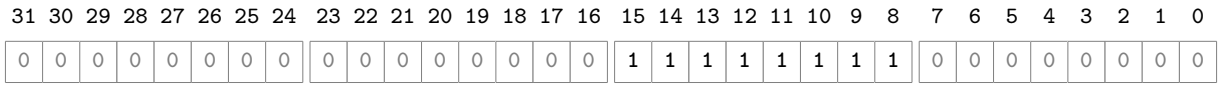
Complete the exercises in-class on your own laptop. You do not need to submit the answers online. Instead, ask an instructor to check your work during the class as soon as you finish answering each question. If you do not finish all the questions in-class, try to finish them outside class and have them checked *at the start of the next exercises class*.

Note: for each exercise you can download a 'template' from Teams that that contains the test program shown in this document.

Bit-wise manipulation of integers

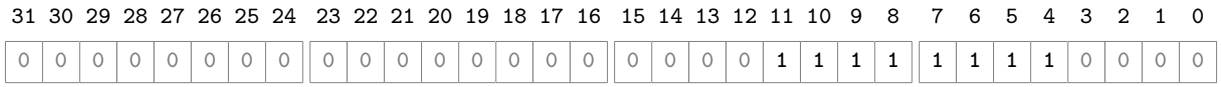
In an integer, bits are numbered from 0 (the least significant, right-most bit) up to 31 (the most significant, left-most bit).

`0b00000000000000001111111100000000 ==`

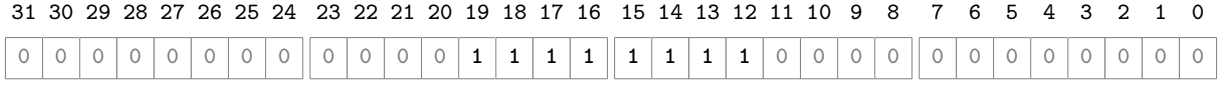


The bits in an integer can be shifted left with the '<<' operator and right with the '>>' operator.

`0b00000000000000001111111100000000 >> 4 ==`
`0b00000000000000000000111111110000`



`0b00000000000000001111111100000000 << 4 ==`
`0b00000000000011111111000000000000`



Bit-wise operations are performed in parallel on each bit of an integer. Complement '`~i`' inverts each bit in `i`. The result of…

bit-wise and '`i&j`' is 1 wherever **both `i` and `j`** have a 1 in the same position;
bit-wise or '`i|j`' is 1 wherever **either `i` or `j`** have a 1 in that position;
exclusive-or '`i^j`' is 1 wherever corresponding bits in `i` and `j` are different.

The following patterns are especially useful for manipulating bits.

```
(1 << N)          // integer with only bit N set to 1
(1 << N) - 1      // integer with the least significant N bits set to 1
~i                // inverts ('flips') all bits in i
i &   mask        // copies bits from i wherever mask has a 1
i &= ~mask;       // sets bits in i to 0 wherever mask has a 1
i |=  mask;       // sets bits in i to 1 wherever mask has a 1
i ^=  mask;       // inverts bits in i wherever mask has a 1
i &   (1 << N)    // tests bit N in i (i.e., 'true' if bit N is 1)
i &= ~(1 << N);   // sets bit number N in i to 0
i |=  (1 << N);   // sets bit number N in i to 1
i ^=  (1 << N);   // inverts ('flips') bit number N in i
(i >> N) & 1      // gets bit N from i (result is 0 or 1)
```

```
(1 << 3)          == 0b1000
(1 << 3) - 1      == 0b0111
              ~0b1100 == 0b0011
0b1100 &  0b0110 == 0b0100
0b1100 & ~0b0110 == 0b1000
0b1100 |  0b0110 == 0b1110
0b1100 ^  0b0110 == 0b1010
```

1 Reading a single bit from an integer

Write a function `int getBit(int i, int p)` that returns the value (0 or 1) of the bit in position `p` of the integer `i`. Hints:

- Move the bit you want (at position `p`) right, to the least significant position, then
- use the bit-wise and operator '&' to isolate just the least significant bit, which is the result you want.

Test your function using this program (download '`05-01-getbit.c`' from Teams):

```
int main()
{
    unsigned int i = 0b10110011100011110000111110000011;

    for (int n = 0;  n < 16;  ++n)
        printf("%u\n", getBit(i, n));

    return 0;
}
```

Verify the output from the program is: `1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0`

2 Setting a single bit in an integer

Write a function `unsigned int setBit(unsigned int i, int p)` that sets the bit at position `p` in the integer `i` to 1.

Test your function with the following program (download '`05-02-setbit.c`' from Teams):

```
int main()
{
    unsigned int i = 0;

    printf("%u\n", i = setBit(i, 1));
    printf("%u\n", i = setBit(i, 3));
    printf("%u\n", i = setBit(i, 5));

    return 0;
```

Verify that the output is: `2 10 42`

3 Clearing a single bit in an integer

Write a function `unsigned int clearBit(unsigned int i, int p)` that sets the bit at position `p` in the integer `i` to 0.

Test your function with the following program (download '`05-03-clearbit.c`' from Teams):

```
int main()
{
    unsigned int i = 62;

    printf("%u\n", i = clearBit(i, 0));
    printf("%u\n", i = clearBit(i, 2));
    printf("%u\n", i = clearBit(i, 4));
```

```
    return 0;
}
```

Verify that the output from the program is: `62 58 42`

4 Getting multiple bits from an integer

Write a function `unsigned int getBits(unsigned int i, int p, int n)` that returns `n` bits from the integer `i`, from bit position `p` upwards, shifted right so that bit `i` is the least-significant bit of the result. For example, to obtain bits 8 through 11 of `0x1234` you would use

```
int result = getBits(0x1234, 8, 4)
```

and the `result` would be `0x0002`. Hints: the easiest way to solve this is in two steps.

- First, shift the integer `i` right so that bit position `p` is moved to the least significant bit (position 0).
- Second, 'mask' off the lowest `n` bits using `(1 << n) - 1` combined with the 'and' operator '`&`'.

Test `getBits()` using the following program (download '`05-04-getbits.c`' from Teams):

```
int main()
{
    unsigned int i = 0b10110011100011110000111110000011;

    printf("%u\n", getBits(i,  0, 8));
    printf("%u\n", getBits(i, 16, 8));
    printf("%u\n", getBits(i, 28, 4));

    return 0;
}
```

Verify that the output from the program is: `131 143 11`

5 Setting multiple bits in an integer

Write a function `unsigned int setBits(unsigned int i, int p, int n, int b)` that sets the `n` bits starting at position `p` in the integer `i` to `b`. Hints:

- A good first step is to extend `clearBit()` so that it can clear `n` bits starting at position `p`.
- When that works, a good second step is to shift `b` left so that its least significant bit moves to the correct position `p`, and then insert it into `i` using the or operator '`|`'.
- Finally, ensure that `b` does not set any bits in `i` outside the range of `n` bits starting at position `p`.

Test your function with the following program (download '`05-05-setbits.c`' from Teams):

```
int main()
{
    unsigned int i = 0b10110011100011110000111110000011;

    printf("%u\n", i = setBits(i,  2, 30,  1)); //   7
    printf("%u\n", i = setBits(i,  6,  2,  7)); // 199
    printf("%u\n", i = setBits(i,  2,  4, 15)); // 255

    return 0;
}
```

(Note that `i` is updated inside each call to `printf()`.) Verify that the output is: `7 199 255`

6 Counting how many bits are set (the number of '1's) in an integer

Write a function `int countBits(unsigned int n)` that counts how many bits in `n` are set (to 1). (For example, `countBits(0b01100110)` is 4.) Hint: One way to do this is to add the least significant bit of `n` to the count, then shift `n` right one bit, until `n` becomes zero.

Test your function using the following program (download '`05-06-countbits.c`' from Teams):

```
int main()
{
    for (int i = 0;  i <= 15;  ++i)
        printf("%d\n", countBits(i));

    return 0;
}
```

Verify that the output is: `0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4`

Manipulating characters in strings

The ASCII encodings of western characters are shown in the table below.

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

One relationship that can be seen is that each lower-case letter encoding is exactly 32 larger than the encoding of the upper-case version of the same letter. (For example, `'D'` is 68 and `'d'` is 68+32 `==` 100.)

7 Converting the case of characters in a string

Write a function `void toLower(char s[])` that converts all upper case letters in `s` to lower case. Hints:

- an upper case character is in the range `'A' <= c <= 'Z'`;

- to convert an upper case character to lower case, add 32 to it;
- the easiest solution is to write a function `int lower(int c)` that returns `c` converted to lower case if it is an upper case character, leaving it unchanged if not, and then overwrite each character in `s` with the result of calling `lower()` on it.

Test your function using the following program (download '`05-07-tolower.c`' from Teams):

```
int main()
{
    char s[] = "Hello World How Are You?";
    toLower(s);
    printf("%s\n", s);
    return 0;
}
```

Verify that the output of the program is: `hello world how are you?`

8 Detecting characters in strings

Write a function `int charIndex(char s[], int c)` that searches `s` for the first occurrence of the character `c`. The result of the function is the index of the first occurrence of `c` in `s`, or `-1` if `c` is not found in `s`.

Test your function with the following program (download '`05-08-charindex.c`' from Teams):

```
int main()
{
    char s[]= "Hello world, how are you?";

    printf("%d\n", charIndex(s, 'H'));
    printf("%d\n", charIndex(s, 'w'));
    printf("%d\n", charIndex(s, '?'));
    printf("%d\n", charIndex(s, 'q'));
    printf("%d\n", charIndex(s, '\0'));

    return 0;
}
```

Verify that the output printed is: `0 6 24 -1 -1`

9 Removing characters from strings

Write a function called `void squeeze(char s[], int c)` that removes all occurrences of the character `c` from the string `s`.

Hint: one way to do this is to copy `s` over itself, using two index variables. One index variable holds the position in `s` of the character currently being tested. The other index variable holds the position in `s` to be overwritten with that character, provided it is not equal to `c`. Don't forget to stop when you reach the 'nul' character at the end of `s`. Also don't forget to terminate `s` again with a 'nul' character, since it might have shrunk.

Test your function using the following program (download '`05-09-squeeze.c`' from Teams):

```c
int main()
{
    char s[]= "Hello world, how are you?";

    squeeze(s, 'o');
    printf("%s\n", s);

    return 0;
}
```

Verify that the result printed is: `Hell wrld, hw are yu?`

10 Removing sets of characters from strings

Write a function `void squeezeAll(char s[], char t[])` that removes all the charcters in `t` from the string `s`. Hint: modify your `squeeze()` function to use your `charIndex()` function to test whether each successive character in `s` is found in `t`.

Test your function with the following program (download '`05-10-squeezeall.c`' from Teams):

```c
int main()
{
    char s[]= "Hello world, how are you?";

    squeezeAll(s, "aeiou");
    printf("%s\n", s);
    squeezeAll(s, " ,");
    printf("%s\n", s);
    squeezeAll(s, "Hld");
    printf("%s\n", s);

    return 0;
}
```

Verify that the output is:

```
Hll wrld, hw r y?
Hllwrldhwry?
wrhwry?
```

11 Challenge: multiple-precision addition [1 bonus point]

There are ten digits in the decimal system. Each position in a decimal number is represented as one of those ten digits with a value between 0 and 9. When adding two decimal numbers `A` and `B` together, the sum in each position `i` is:

$$S_i = (A_i + B_i + C_i) \bmod 10$$

and the carry out of that position into the position to its left is:

$$C_{i+1} = (A_i + B_i) \div 10$$

Similarly, each digit position in a base-256 number has a digit value between 0 and 255. When adding two base-256 numbers together the sum in each position `i` is:

$$S_i = (A_i + B_i + C_i) \bmod 256$$

and the carry out of that position into next more significant position is:

$$C_{i+1} = (A_i + B_i) \div 256$$

Using an array of `unsigned char` you can make a base-256 integer with as many digits as you need, and then implement the usual arithmetic operations for those integers as functions.

Write a function

```
void add(unsigned char a[], unsigned char b[], unigned char s[], int n)
```

that adds two base-256 numbers stored in arrays `a` and `b`, each containing `n` digits, and places the sum in the `n`-digit array `s`. The numbers are stored least significant digit first (at index 0). Make sure your function handles carries correctly. (The initial carry in to digit position 0, C_0, should be 0.)

Test your function using the following program (download '`05-11-bigint.c`' from Teams):

```
int main()
{
    unsigned char a[8] = { 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80 };
    unsigned char b[8] = { 0x90, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0, 0x00 };
    unsigned char s[8];

    add(a, b, s, 8);

    for (int i= 7;  i >= 0;  --i)
        printf("%02x", s[i]);
    printf("\n");

    return 0;
}
```

Verify that the output is: `8161412100e0c0a0`

If that was fun, implement some other operators as functions. If you first implement shift, comparison, and subtraction then implementing multiplication and division becomes relatively straightforward.