2_1 Information Storage

- Rather than accessing individual bits in memory, most computers use blocks of 8 bits, or bytes, as the smallest addressable unit of memory.
- A machine-level program views memory as a very large array of bytes, referred to as virtual memory. Every byte of memory is identified by a unique number, known as its address, and the set of all possible addresses is known as the virtual address space.
    - As indicated by its name, this virtual address space is just a conceptual image presented to the machine-level program.
- In subsequent chapters, we will cover how the compiler and run-time system partitions **this memory space into more manageable units to store the different program objects**, that is, program data, instructions, and control information.
- Various mechanisms are used to allocate and manage the storage for different parts of the program. This management is all performed **within the virtual address space.**
- Example (C pointer):
    - Value of a pointer in C - the virtual address of the first byte of some block of storage, no matter whether it points to an integer, a structure, or some other program object.
    - Type of a pointer in C - C compiler also associates type information with each pointer, so that it can generate different machine-level code to access the value stored at the location designated by the pointer depending on the type of that value.

    ```
    int a[10]; // a's value is a virtual address. Type int: a+4
    char b[10]; // b's value is also a virtual address. Type char: b+1
    ```

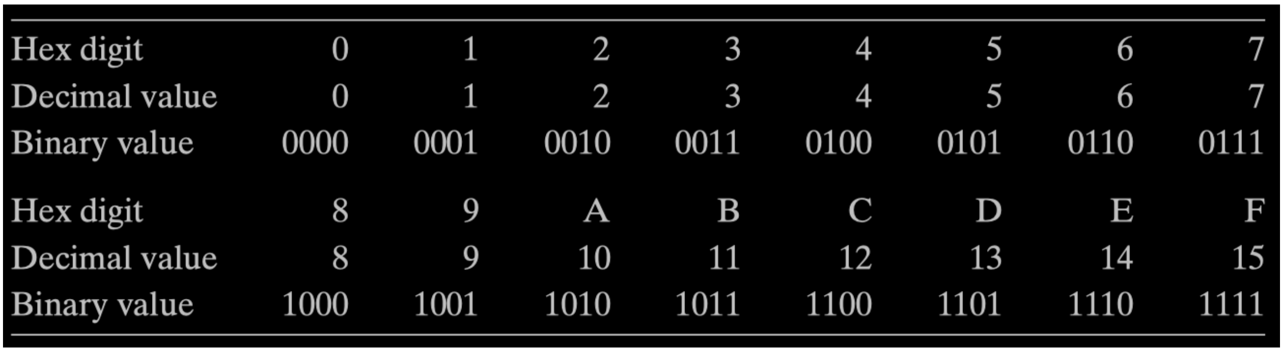
    - Although the C compiler maintains this type information, the actual machine-level program it generates has no information about data types. It simply treats each program object as a block of bytes and the program itself as a sequence of bytes.

2.1.1 Hexadecimal Notation

- Reason for hexadecimal notation, a single byte consists of 8 bits.
    - In binary notation, its value ranges from 0B 00000000 to 0B 11111111 - too complex.
    - In decimal, its value ranges from 0 to 255. - No bit patterns' description.
    - Neither notation is very convenient for describing bit patterns.
    - Hexadecimal notation can be simple and better descriptive of bit patterns at the same time.
- Structure of hexadecimal
    - Hexadecimal (or simply “hex”) uses digits ‘0’ through ‘9’ along with characters ‘A’ through ‘F’ to represent 16 possible values.
    - In C, numeric constants starting with 0x or 0X are interpreted as being in hexadecimal.
    - The characters ‘A’ through ‘F’ may be written in either upper- or lowercase.

```
0x1FA6 or 0x1fa6
```

- A common task in working with machine-level programs is to manually convert between decimal, binary, and hexadecimal representations of bit patterns:



Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

- Example of converting hex to binary:

```
Hex: 0x 567abe4f
Binary: 0b 0101 0110 0111 1010 1011 1110 0100 1111
```

- Some tricks: $x = 2^n$
  - Suppose: n = i + 4*j
  - Then we can write x with a leading hex digit of 1 (i = 0), 2 (i = 1), 4 (i = 2), or 8 (i = 3), followed by j hexadecimal 0s.
  - Example:
    - 2^{11} -> 11 = 3 + 4*2 -> i=3,j=2 -> 0x800
    - 2^{45} -> 45 = 1 + 4*11 -> i=1,j=11 -> 0x200000000000

Practice Problem 2.1

Perform the following number conversions:

A. 0x25B9D2 to binary
B. binary 1010111001001001 to hexadecimal
C. 0xA8B3D to binary
D. binary 1100100010110110010110 to hexadecimal

Solution:

A. 0x25B9D2 to binary

```
0x25B9D2 = 0b 0010 0101 1011 1001 1101 0010
```

B. binary 1010111001001001 to hexadecimal

```
0b 1010 1110 0100 1001 = 0xAE49
```

C. 0xA8B3D to binary

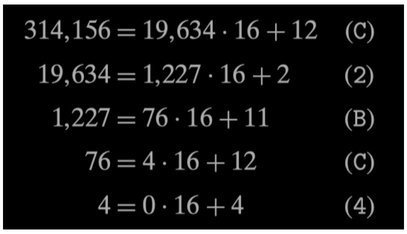
```
0xA8B3D = 0b 1010 1000 1011 0011 1101
```

D. binary 1100100010110110010110 to hexadecimal

```
0b 11 0010 0010 1101 1001 0110 = 0x322D96
```

- Converting between decimal and hexadecimal representations requires using multiplication or division to handle the general case.
- Decimal -> hexadecimal representations:
  - To convert a decimal number x to hexadecimal, we can repeatedly divide x by 16, giving a quotient q and a remainder r, such that x = q . 16 + r.

- We then use the hexadecimal digit representing r as the least significant digit and generate the remaining digits by repeating the process on q.



$$\begin{aligned} 314{,}156 &= 19{,}634 \cdot 16 + 12 & (C) \\ 19{,}634 &= 1{,}227 \cdot 16 + 2 & (2) \\ 1{,}227 &= 76 \cdot 16 + 11 & (B) \\ 76 &= 4 \cdot 16 + 12 & (C) \\ 4 &= 0 \cdot 16 + 4 & (4) \end{aligned}$$

314156 = 0x 4cb2c

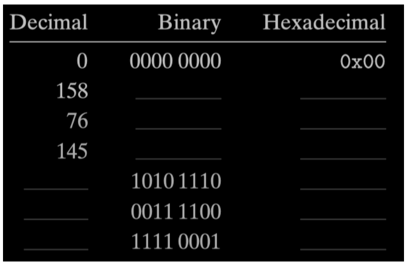
```
For decimal: 500000, how to get its hex value?
500000 = 31250 * 16 + 0 (0)
31250 = 1953 * 16 + 2   (2)
1953 = 122 * 16 + 1             (1)
122 = 7 * 16 + 10               (A)
7                                                                               (7)

decimal 500000 = hex 0x 7a120
```

```c
// decToHex.c
// Require customer to input a decimal, then program will translate to hex and print.

#include<stdio.h>

int main()
{
        int i_inputNum; // store the input number
        char alphabet[0x10] = "0123456789ABCDEF";       // alphabet table for hex
        char result[0x100];
        int length;     // the length of result array

        printf("This program will convert a decimal you input to a hex accordinglly.\n");
        printf("Please input a decimal number:");
        scanf("%d",&i_inputNum);

        if(i_inputNum == 0) printf("0x0");

        // Calculate the result and store into result array
        for(int i = 0;i_inputNum!=0;i++)
        {
                result[i] = alphabet[i_inputNum%0x10];// repeatly divide input number by 16
                                                      // record the remainder into result[i]
                i_inputNum /= 0x10;     // get the quotient
                length = i;
        }

        // Print the result
        printf("The hex result: 0x");
        for(int i = length; i >= 0; i--) printf("%c",result[i]);
        printf("\n");

        return 0;
}

```

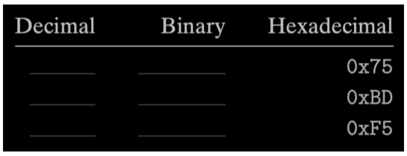
- Hexadecimal -> decimal representations:
  - Multiply each of the hexadecimal digits by the appropriate power of 16.
  - Example:
    - hex: 0x8AF
    - dec: $8 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 1967$

Practice Problem 2.3

A single byte can be represented by 2 hexadecimal digits. Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns:



Decimal	Binary	Hexadecimal
0	0000 0000	0x00
158		
76		
145		
	1010 1110	
	0011 1100	
	1111 0001	



Decimal	Binary	Hexadecimal
		0x75
		0xBD
		0xF5

Solution:

```
Decimal                         Binary                                  Hexadecimal
158                                         0b 1110 1001                               0x E9
76                                          0b 1100 0100                               0x C4
145                                         0b 0001 1001                               0x 19
174                                         0b 1010 1110                               0x AE
60                                          0b 0011 1100                               0x 3C
241                                         0b 1111 0001                               0x F1
```

```
Decimal                                 Binary                      Hex
117                                                 0b 0111 0101    0x75
189                                                 0b 1011 1101    0xBD
245                                                 0b 1111 0101    0xF5
```

Practice Problem 2.4

Without converting the numbers to decimal or binary, try to solve the following arithmetic problems, giving the answers in hexadecimal. Hint: Just modify the methods you use for performing decimal addition and subtraction to use base 16.

A. 0x605c + 0x5 = ?

B. 0x605c − 0x20 = ?

C. 0x605c + 32 = ?

D. 0x60fa − 0x605c = ?

Solution:

A. 0x605c + 0x5 = 0x6062

B. 0x605c − 0x20 = 0x603c

C. 0x605c + 32(0x20) = 0x607c

D. 0x60fa − 0x605c = 0x 9e

2.1.2 Data Sizes

- Word Size:
  - Every computer has a word size, indicating the nominal size of pointer data.
  - Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is **the maximum size of the virtual address space.**

```
suppose word size is 8 bits
```

```
virtual address space range: 00000000 ~ 11111111
```

  - For a machine with a w-bit word size (32位或64位系统), the virtual addresses can range from 0 to $2^w - 1$, giving the program access to at most 2^w bytes.
- 32bit and 64bit:
  - 32-bit word size - 4 gigabytes
  - 64-bit word size - 16 exabytes
- Compile:

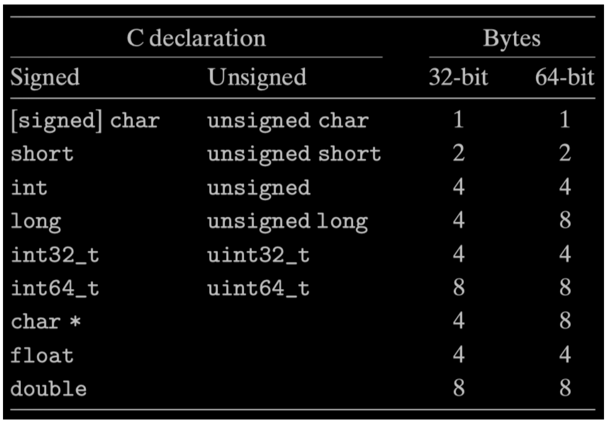
```
linux> gcc -m32 prog.c
```

  - This program will run correctly on either a 32-bit or a 64-bit machine.

```
linux> gcc -m64 prog.c
```

  - This program will only run on a 64-bit machine.

- We will therefore refer to programs as being either “32-bit programs” or “64-bit programs,” since the distinction lies in **how a program is compiled**, rather than the type of machine on which it runs.
- Computers and compilers support multiple data formats using different ways to encode data, such as integers and floating point, as well as different lengths.
- The C language supports multiple data formats for both integer and floating-point data.



C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

  - To avoid the vagaries of relying on “typical” sizes and different compiler settings, ISO C99 introduced a class of data types where the data sizes are fixed regardless of compiler and machine settings. Among these are data types int32_t and int64_t, having **exactly 4 and 8 bytes**.
  - Most of the data types encode signed values, unless prefixed by the keyword unsigned or using the specific unsigned declaration for fixed-sized data types.

```
    unsigned int
```

- The C language allows a variety of ways to order the keywords and to include or omit optional keywords.

```
unsigned long
unsigned long int
long unsigned
long unsigned int
```

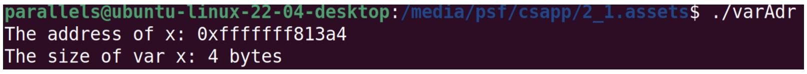
  - A pointer uses the **full word size** of the program.
  - Programmers should strive to make their programs portable across different machines and compilers. One aspect of portability is to make the program insensitive to the

exact sizes of the different data types.

2.1.3 Addressing and Byte Ordering (寻址和字节顺序)

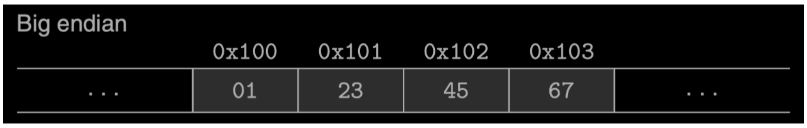
- In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used.
- Example:
    - A variable x of type int has address 0x100. So the value of &x is 0x100, which is address of variable x.
    - Code:

```c
// varAdr.c
int x = 5;
printf("The address of variable x is: %p \n", &x)
```



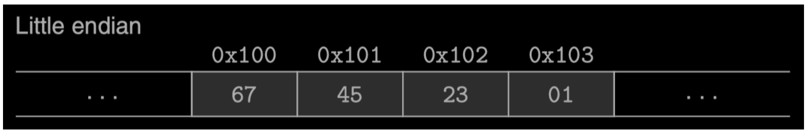
```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2_1.assets$ ./varAdr
The address of x: 0xfffffff813a4
The size of var x: 4 bytes
```

- 2 common conventions of ordering the bytes representing an object:
    - little endian(小端排序) - the least significant byte(最低有效字节) comes first from lower address
    - big endian(大端排序) - the most significant byte (最高有效字节) comes first from lower address
    - Example:
        - For variable x of type in and at address 0x100 has a hexadecimal value of 0x01234567



Big endian

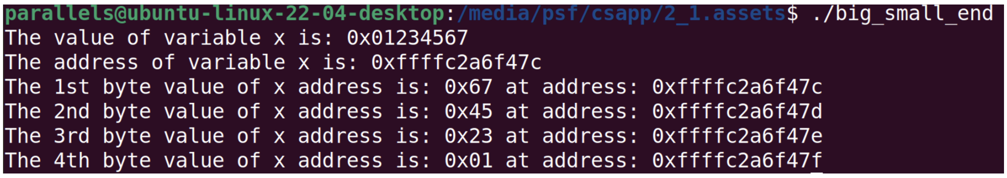
	0x100	0x101	0x102	0x103	
...	01	23	45	67	...



Little endian

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

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

int main()
{
        uint32_t x = 0x01234567;
        uint32_t* x_adr = &x;
        printf("The value of variable x is: 0x%08x \n",x);
        printf("The address of variable x is: %p\n",x_adr);
        printf("The 1st byte value of x address is: 0x%02x \n",*(char*)x_adr);
        printf("The 2nd byte value of x address is: 0x%02x \n",*((char*)x_adr+1));
        printf("The 3rd byte value of x address is: 0x%02x \n",*((char*)x_adr+2));
        printf("The 4th byte value of x address is: 0x%02x \n",*((char*)x_adr+3));
        return 0;
}
```



```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2_1.assets$ ./big_small_end
The value of variable x is: 0x01234567
The address of variable x is: 0xffffc2a6f47c
The 1st byte value of x address is: 0x67 at address: 0xffffc2a6f47c
The 2nd byte value of x address is: 0x45 at address: 0xffffc2a6f47d
The 3rd byte value of x address is: 0x23 at address: 0xffffc2a6f47e
The 4th byte value of x address is: 0x01 at address: 0xffffc2a6f47f
```

- Most Intel-compatible machines operate exclusively in little-endian mode. On the other hand, most machines from IBM and Oracle (arising from their acquisition of Sun Microsystems in 2010) operate in big-endian mode.
- Possible issues of differences on little-endian and big-endian mode:
  1. Binary data are communicated over a network between different machines.
     - To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard.
  2. When looking at the byte sequences representing integer data, which occurs often when inspecting machine-level programs.
     - Example:

```
4004d3: 01 05 43 0b 20 00                       add %eax,0x200b43(%rip)
```

     - eax -> (0x200b43 + rip)
     - Operation: adds a word of data to the value stored at an address computed by adding 0x200b43 to the current value of the program counter.

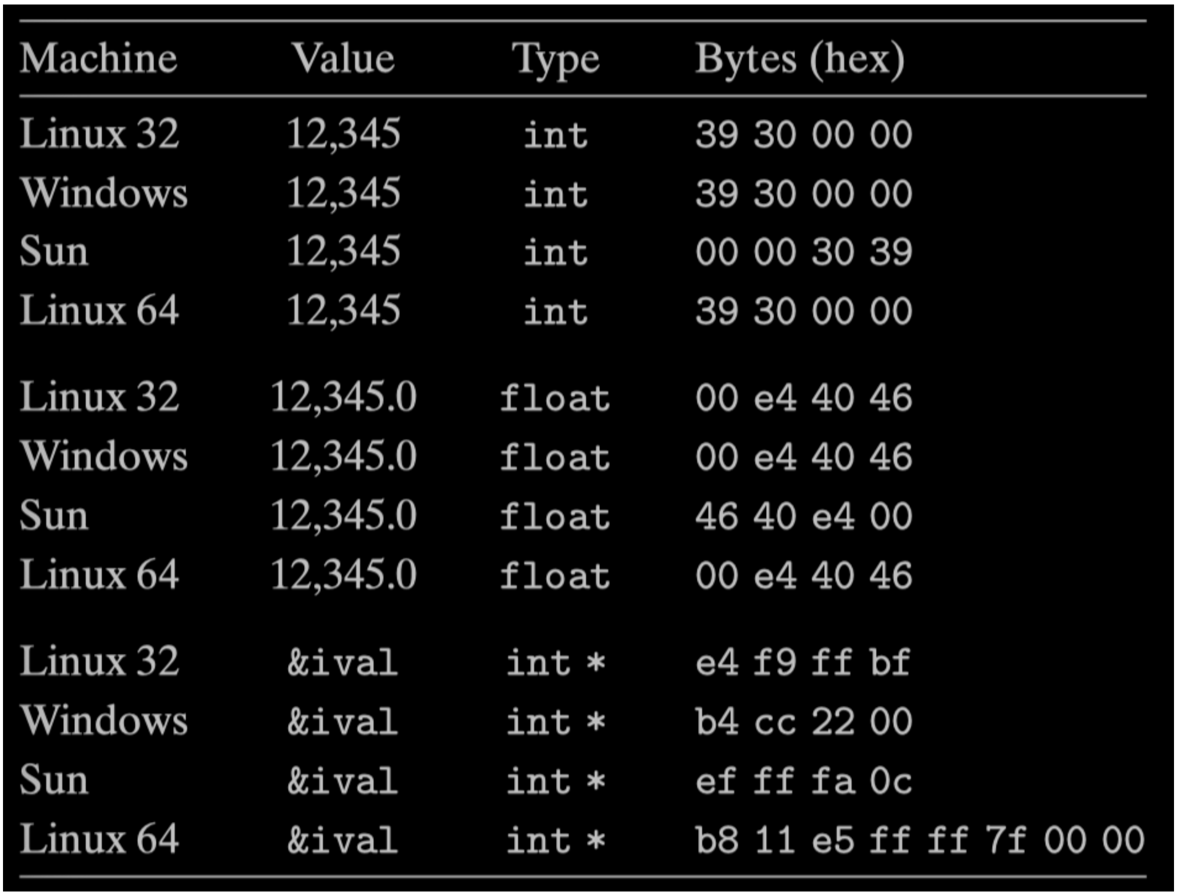
```
Final 4 bytes:
4004d5: 43 0b 20 00
If little-endian, the value is: 0x 00 20 0b 43
If big-endian, the value is:            0x 43 0b 20 00
```

  3. Using a cast(强制类型转换) or a union to allow an object to be referenced according to a different data type from which it was created.

```c
// cast.c
#include <stdio.h>

typedef unsigned char* byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
        int i;
        for(i = 0; i < len; i++) printf(" %.2x", start[i]);
        printf("\n");
}

void show_int(int x) {
        show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
        show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
        show_bytes((byte_pointer) &x, sizeof(void *));
}

void test_show_bytes(int val) {
  int ival      = val;
  float fval = (float) ival;
        int     *pval   =       &ival;
        show_int(ival);
  show_float(fval);
  show_pointer(pval);
}
```

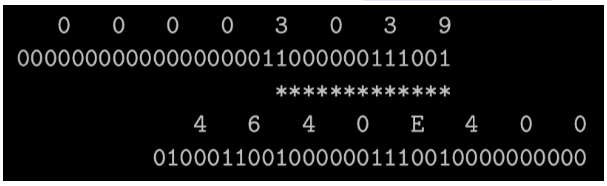
     - show_bytes is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count.
     - show_int, show_float, and show_pointer demonstrate how to use procedure show_bytes to print the byte representations of C program objects of type int,

float, and void*.



Machine	Value	Type	Bytes (hex)
Linux 32	12,345	int	39 30 00 00
Windows	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Linux 64	12,345	int	39 30 00 00
Linux 32	12,345.0	float	00 e4 40 46
Windows	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Linux 64	12,345.0	float	00 e4 40 46
Linux 32	&ival	int *	e4 f9 ff bf
Windows	&ival	int *	b4 cc 22 00
Sun	&ival	int *	ef ff fa 0c
Linux 64	&ival	int *	b8 11 e5 ff ff 7f 00 00

- Decimal: 12345 = Hex: 0x 00003039
  - Linux 32, Windows, and Linux 64, indicating little-endian machines.
  - Sun, indicating a big-endian machine.
- Float: 12345.0 = Hex: 0x 4640E400
  - the bytes of the float data are identical, except for the byte ordering.
- Pointer: &ival -> totally different.
  - Linux 32, Windows, and Sun machines use 4-byte addresses, while the Linux 64 machine uses 8-byte addresses.
- Observations:
  - the floating-point and the integer data both encode the numeric value 12,345, they have very different byte patterns and use different encoding schemes.



```
   0   0   0   0   3   0   3   9
00000000000000000011000000111001
                   *************
             4   6   4   0   E   4   0   0
          01000110010000001110010000000000
```

Practice Problem 2.5

Consider the following three calls to show_bytes:

```
int a = 0x12345678;
byte_pointer ap = (byte_pointer) &a;
show_bytes(ap, 1); /* A. */
show_bytes(ap, 2); /* B. */
show_bytes(ap, 3); /* C. */
```

Indicate the values that will be printed by each call on a little-endian machine

A. Little endian: ? Big endian: ?

B. Little endian: ? Big endian: ?

C. Little endian: ? Big endian: ?

Solution:

A. Little endian: 0x 78 Big endian: 0x 12

B. Little endian: 0x 78 56 Big endian: 0x 12 34

C. Little endian: 0x 78 56 34 Big endian: 0x 12 34 56

```c
//cast.c
#include <stdio.h>

int main()
{
        int a = 0x12345678;

        byte_pointer ap = (byte_pointer) &a;

        show_bytes(ap, 1); /* A. */

        show_bytes(ap, 2); /* B. */

        show_bytes(ap, 3); /* C. */

        return 0;
}
```

Practice Problem 2.6

Using show_int and show_float, we determine that the integer 2607352 has hexadecimal representation 0x0027C8F8, while the floating-point number 2607352.0(original wrong point) has hexadecimal representation 0x4A1F23E0.

A. Write the binary representations of these two hexadecimal values.
B. Shift these two strings relative to one another to maximize the number of matching bits. How many bits match?
C. What parts of the strings do not match?

Solution:
A. Write the binary representations of these two hexadecimal values.

Show hex code:

```c
// P2_6.c
#include <stdio.h>
#include <string.h>

typedef unsigned char* byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
        int i;
        printf("Show %ld bytes from address: %p\n",len,start);
        for(i = 0; i < len; i++)
                printf("\t%p: %.2x\n",start+i, start[i]);
        printf("\n");
}

void show_int(int x) {
        show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
        show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
        show_bytes((byte_pointer) &x, sizeof(void *));
}

int main()
{
        char str_type[0x10] = {0};
        printf("Please choose the kind of conversion (int/float/pointer): ");
        scanf("%s",str_type);
        if(!strcmp(str_type,"int"))
        {
                int tgt_int;
                printf("Please input the target integer:");
                scanf("%d",&tgt_int);
                show_int(tgt_int);
        }
        else if(!strcmp(str_type,"float"))
        {
                float tgt_float;
                printf("Please input the target float:");
                scanf("%f",&tgt_float);
                show_float(tgt_float);
        }
        else if(!strcmp(str_type,"pointer"))
        {
                int val = 0;
                printf("Please input a number then its pointer will be calculated:");
                scanf("%d",&val);
                void* tgt_pointer = (void*)&val;
                show_pointer(tgt_pointer);
        }
        else
        {
                printf("Wrong inputs!\n");
        }
        return 0;
}
```

```
Decimal: 2607352 Hex: 0x0027C8F8
0b 0000 0000 0010 0111 1100 1000 1111 1000
Float: 2607352.0 Hex: 0x4a1f23e0
0b 0100 1010 0001 1111 0010 0011 1110 0000
```

B. Shift these two strings relative to one another to maximize the number of matching bits. How many bits match?

```
Decimal: 2607352 Hex: 0x0027C8F8
0b 00000000001001111100100011111000
Float: 2607352.0 Hex: 0x4a1f23e0
0b   01001010000111110010001111100000

21 bits match.
```

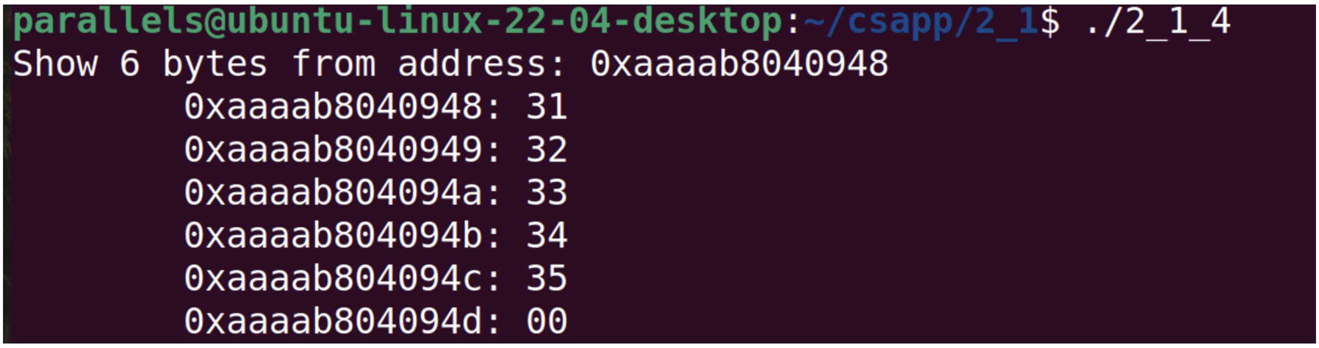
C. What parts of the strings do not match?

Except the highest bit, the other bits of integer are included in the float bits.

2.1.4 Representing Strings

- A string in C is encoded by an array of characters terminated by the **null** (having value 0) character.
- Example:

```c
show_bytes("12345",6);
```



```
parallels@ubuntu-linux-22-04-desktop:~/csapp/2_1$ ./2_1_4
Show 6 bytes from address: 0xaaaab8040948
        0xaaaab8040948: 31
        0xaaaab8040949: 32
        0xaaaab804094a: 33
        0xaaaab804094b: 34
        0xaaaab804094c: 35
        0xaaaab804094d: 00
```

- the ASCII code for decimal digit x happens to be 0x3x, and that the terminating byte has the hex representation 0x00.

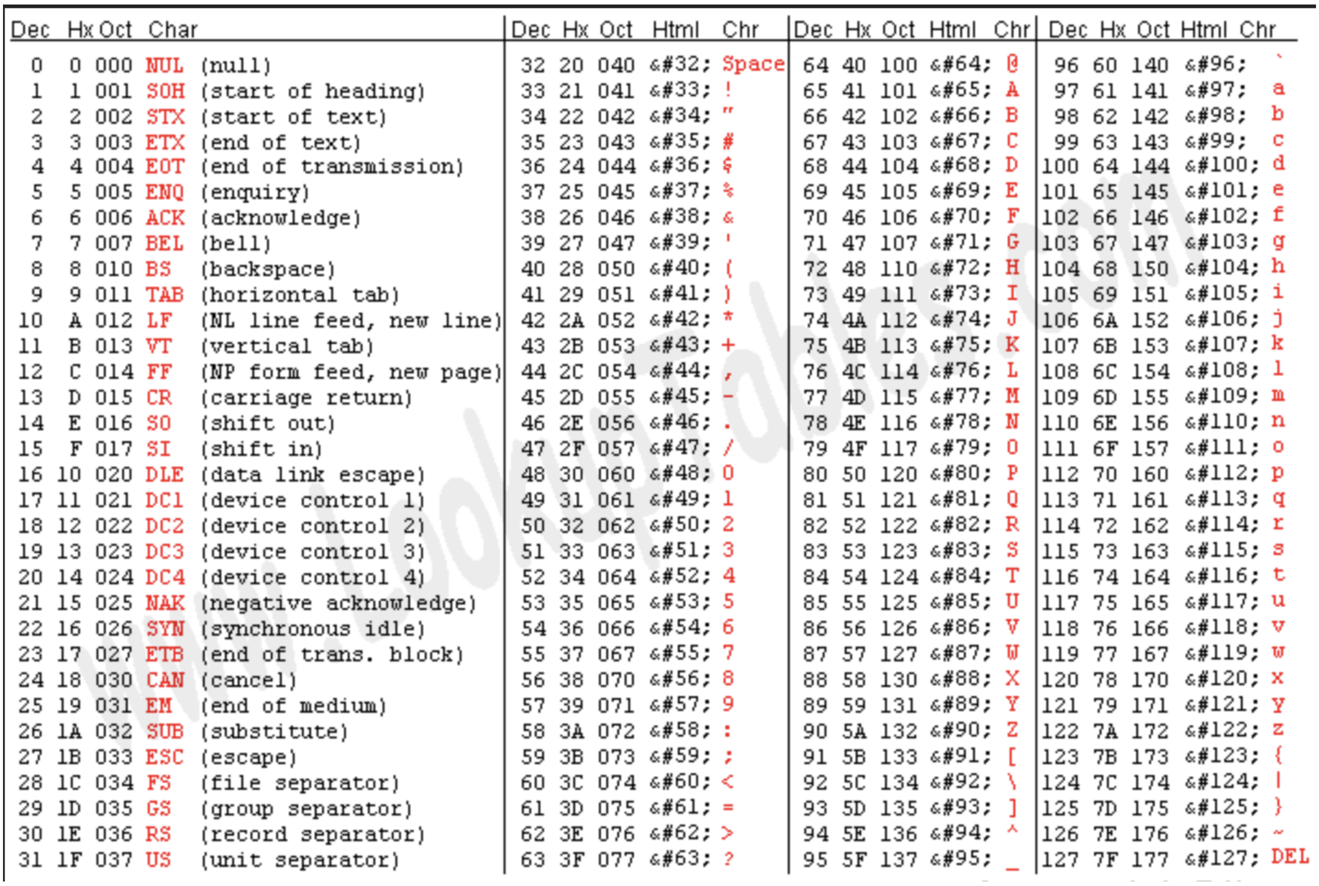
Practice Problem 2.7

What would be printed as a result of the following call to show_bytes?

```
const char *m = "mnopqr";
show_bytes((byte_pointer) m, strlen(m));
```

Note that letters ‘a’ through ‘z’ have ASCII codes 0x61 through 0x7A.

Solution:



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

prints 6D 6E 6F 70 71 72.

2.1.5 Representing Code

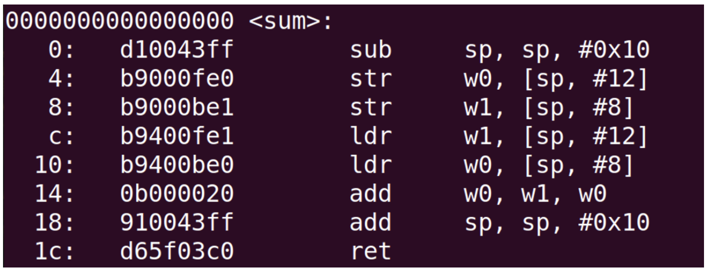
- Example:

```c
int sum(int x, int y) {
        return x + y;
}
```

- how to check its byte representations:

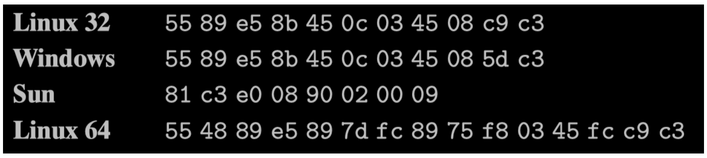
  ```
  $ gcc -og -c main.c -o main.o
  $ objdump -d main.o
  ```

  - arm cpu:

    ```
    0000000000000000 <sum>:
       0:   d10043ff        sub     sp, sp, #0x10
       4:   b9000fe0        str     w0, [sp, #12]
       8:   b9000be1        str     w1, [sp, #8]
       c:   b9400fe1        ldr     w1, [sp, #12]
      10:   b9400be0        ldr     w0, [sp, #8]
      14:   0b000020        add     w0, w1, w0
      18:   910043ff        add     sp, sp, #0x10
      1c:   d65f03c0        ret
    ```

  - X86 cpu:

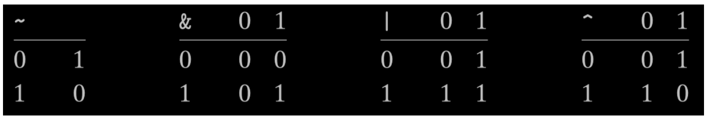
    

Linux 32	55 89 e5 8b 45 0c 03 45 08 c9 c3
Windows	55 89 e5 8b 45 0c 03 45 08 5d c3
Sun	81 c3 e0 08 90 02 00 09
Linux 64	55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

- Highlights:
  - Instruction codings are different. Different machine types use different and incompatible instructions and encodings. Even identical processors running different operating systems have differences in their coding conventions and hence are not binary compatible.
- **Binary code is seldom portable across different combinations of machine and operating system.**
- A program, from the perspective of the machine, is simply a sequence of bytes.
- The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging.

2.1.6 Introduction to Boolean Algebra

- Boolean algebra - logic values true and false as binary values 1 and 0.
- Operations of boolean algebra - NOT, AND, OR, EXCLUSIVE-OR:



~	
0	1
1	0

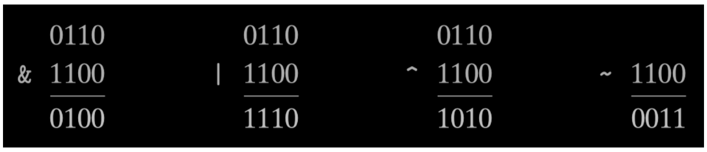
&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

  - ~ corresponds to the logical operation not, denoted by the symbol ¬ (命题逻辑). That is, we say that ¬P is true when P is not true, and vice versa. ~p equals 1 when p equals 0, and vice versa.
  - & corresponds to the logical operation and, denoted by the symbol ∧. We say that P ∧ Q holds when both P is true and Q is true. Correspondingly, p & q equals 1 only when p = 1 and q = 1.
  - | corresponds to the logical operation or, denoted by the symbol ∨. We say that P ∨ Q holds when either P is true or Q is true. Correspondingly, p | q equals 1 when either p = 1 or q = 1.
  - ^ corresponds to the logical operation exclusive-or, denoted by the symbol ⊕ . We say that P ⊕ Q holds when either P is true or Q is true, but not both. Correspondingly, p ^ q equals 1 when either p = 1 and q = 0, or p = 0 and q = 1.

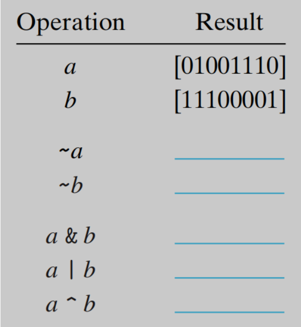
- We can extend the four Boolean operations to also operate on bit vectors(位向量运算), strings of zeros and ones of some fixed length w.
- Example:
  - Let a and b denote the bit vectors $[a_{w-1}, a_{w-2}, \ldots, a_0]]$ and $[b_{w-1}, b_{w-2}, \ldots, b_0]$, respectively.
  - We define a & b to also be a bit vector of length w, where the i-th element equals a i& b i, for 0 ≤ i < w.
  - Suppose w=4,a= [0110],b=[1100]:



```
    0110        0110        0110
&   1100    |   1100    ^   1100    ~   1100
    0100        1110        1010        0011
```

Practice Problem 2.8

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.



Operation	Result
a	[01001110]
b	[11100001]
~*a*	________
~*b*	________
a & *b*	________
a | *b*	________
a ^ *b*	________

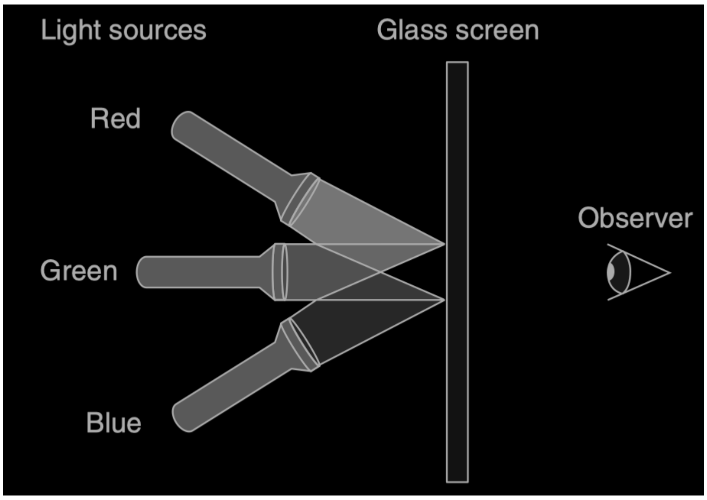
Solution:

```
a       = [01001110]
b       = [11100001]
~a      = [10110001]
~b      = [00011110]
a&b = [01000000]
a|b = [11101111]
a^b = [10101111]
```

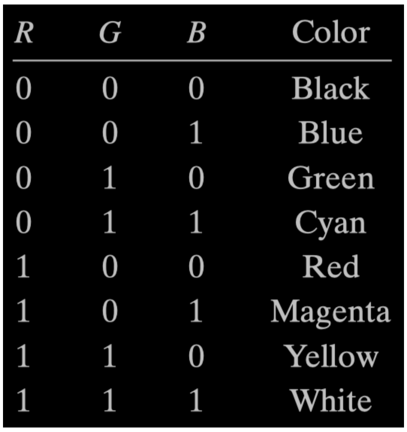
- One useful application of bit vectors is to represent finite sets（有限集合）.
  - We can encode any subset A ⊆ { 0, 1, . . . , w − 1 } with a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where a i = 1 if and only if i ∈ A.
  - Example:
    - bit vector a = [01101001]encodes the set A = { 0, 3, 5, 6 }
    - bit vector b = [01010101]encodes the set B = { 0, 2, 4, 6 }
  - operations | and & correspond to set union(并) and intersection(交).
  - ~ corresponds to set complement（补）.
  - Example:
    a = [01001110] -> A = {1,2,3,6}
    b = [11100001] -> B = {0,5,6,7}
    ~a = [10110001] -> $\overline{A}$ = {0,4,5,7}
    ~b = [00011110] -> $\overline{B}$ = {1,2,3,4}
    a&b = [01000000] -> A∩B = { 6 }
    a|b = [11101111] -> A∪B = {0,1,2,3,5,6,7}
    a^b = [10101111] -> A⊕B = {0,1,2,3,5,7}

Practice Problem 2.9

Computers generate color pictures on a video screen or liquid crystal display by mixing three different colors of light: red, green, and blue. Imagine a simple scheme, with three different lights, each of which can be turned on or off, projecting onto a glass screen:



We can then create eight different colors based on the absence (0) or presence (1) of light sources R, G, and B:



R	*G*	*B*	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Each of these colors can be represented as a bit vector of length 3, and we can apply Boolean operations to them.

A. The complement of a color is formed by turning off the lights that are on and turning on the lights that are off. What would be the complement of each of the eight colors listed above?

B. Describe the effect of applying Boolean operations on the following colors:

Blue | Green = ?

Yellow & Cyan = ?

Red ^ Magenta = ?

Solution:

A.

```
black = [0,0,0] -> ~black = [1,1,1] = white
blue  = [0,0,1] -> ~blue = [1,1,0] = yellow
green = [0,1,0] -> ~green = [1,0,1] = magenta
cyan(青色)  = [0,1,1] -> ~cyan [1,0,0] = red
red   = [1,0,0] -> ~red = cyan
magenta(洋红) = [1,0,1] -> ~magenta = green
yellow = [1,1,0] -> ~yellow = blue
white  = [1,1,1] -> ~white = black
```

B.

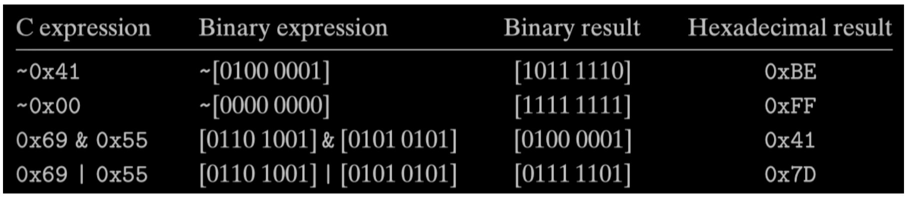
```
Blue | Green = [001]|[010] = [011] = Cyan

Yellow & Cyan = [110]&[011] = [010] = Green

```

```
Red ^ Magenta = [100]^[101] = [001] = Blue
```

2.1.7 Bit-Level Operations in C

- C supports bitwise Boolean operations.

```
| for or         // 或
& for and        // 与
~ for not        // 取反
^ for exclusive-or      // 异或
```



C expression	Binary expression	Binary result	Hexadecimal result
~0x41	~[0100 0001]	[1011 1110]	0xBE
~0x00	~[0000 0000]	[1111 1111]	0xFF
0x69 & 0x55	[0110 1001] & [0101 0101]	[0100 0001]	0x41
0x69 | 0x55	[0110 1001] | [0101 0101]	[0111 1101]	0x7D

- The best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

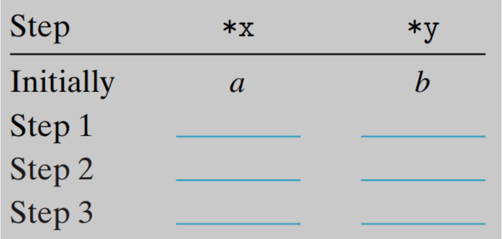
Practice Problem 2.10

As an application of the property that a ^ a = 0 for any bit vector a, consider the following program:

```c
void inplace_swap(int *x, int *y) {
        *y = *x ^ *y;   // Step 1
        *x = *x ^ *y;   // Step 2
        *y = *x ^ *y;   // Step 3
}
```

As the name implies, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables x and y. Note
that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is
no performance advantage to this way of swapping; it is merely an intellectual amusement.

Starting with values a and b in the locations pointed to by x and y, respectively, fill in the table that follows, giving the values stored at the two locations after each step of the procedure. Use the properties of ^ to show that the desired effect is achieved. Recall that every element is its own additive inverse (that is, a ^ a = 0).



Step	*x	*y
Initially	*a*	*b*
Step 1	________	________
Step 2	________	________
Step 3	________	________

Solution:

- Purpose: swap the values stored at the locations denoted by pointer variables x and y.
- Do not need a third location to temporarily store one value while we are moving the other.

```
Step               *x     *y
Initially     a      b
```

```
Step1              a     a^b
Step2              b         a^b
Step3              b     a
```

Practice Problem 2.11

Armed with the function `inplace_swap` from Problem 2.10, you decide to write code that will reverse the elements of an array by swapping elements from opposite ends of the array, working toward the middle.

You arrive at the following function:

```
void reverse_array(int a[], int cnt) {
        int first, last;
        for(first = 0, last = cnt - 1; first <= last;first++,last--)
    inplace_swap(&a[first], &a[last]);
}
```

When you apply your function to an array containing elements 1, 2, 3, and 4, you find the array now has, as expected, elements 4, 3, 2, and 1. When you try it on an array with elements 1, 2, 3, 4, and 5, however, you are surprised to see that the array now has elements 5, 4, 0, 2, and 1. In fact, you discover that the code always works correctly on arrays of even length, but it sets the middle element to 0 whenever the array has odd length.

A. For an array of odd length cnt = 2k + 1, what are the values of variables first and last in the final iteration of function reverse_array?

B. Why does this call to function `inplace_swap` set the array element to 0?

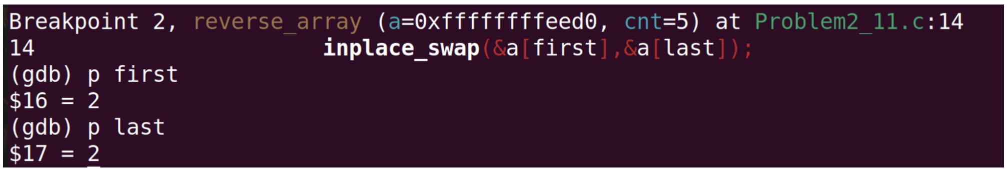
C. What simple modification to the code for reverse_array would eliminate this problem?

Solution:

```
1    #include<stdio.h>
2
3    void inplace_swap(int* x, int* y)
4    {
5            *y = *x ^ *y;
6            *x = *x ^ *y;
7            *y = *x ^ *y;
8    }
9
10   void reverse_array(int a[], int cnt)
11   {
12           int first, last;
13           for(first = 0, last = cnt - 1; first <= last; first++, last--)
14                   inplace_swap(&a[first],&a[last]);
15   }
16
17
18   int main()
19   {
20           int test[] = {0,1,2,3,4};
21           reverse_array(test,5);
22           for(int i = 0; i < sizeof(test)/sizeof(test[0]);i++)
23                   printf("%d ",test[i]);
24           return 0;
25   }
```

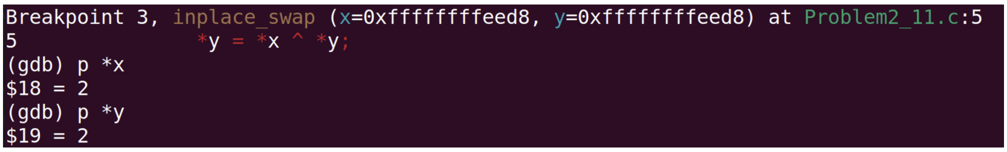
```
gdb debugging:
gcc -g test.c -o test   // add debugging symbols to test program
gdb test        // start gdb to debug test program
l 1,26  // list the code from line 1 to 26
b 14            // add breakpoint on line 14
r                               // start run the code from first line
```

```
p test[2]       // check variable content while hit breakpoint
```



```
Breakpoint 2, reverse_array (a=0xffffffffeed0, cnt=5) at Problem2_11.c:14
14                      inplace_swap(&a[first],&a[last]);
(gdb) p first
$16 = 2
(gdb) p last
$17 = 2
```

B.



```
Breakpoint 3, inplace_swap (x=0xffffffffeed8, y=0xffffffffeed8) at Problem2_11.c:5
5                 *y = *x ^ *y;
(gdb) p *x
$18 = 2
(gdb) p *y
$19 = 2
```

So *x ^ *y = 0, which make it wrong result!

C.

first <= last ->first < last

```
void reverse_array(int a[], int cnt)
{
        int first, last;
        for(first = 0, last = cnt - 1; first < last; first++, last--)
              inplace_swap(&a[first],&a[last]);
}
```

- One common use of bit-level operations is to implement masking operations, where a mask is a bit pattern that indicates a selected set of bits within a word.
    - The mask 0xFF indicates the low-order byte of a word. The bit-level operation x & 0xFF yields a value consisting of the least significant byte of x, but with all other bytes set to 0.
    - with x = 0x89ABCDEF, the expression would yield 0x000000EF.

Practice Problem 2.12

Write C expressions, in terms of variable x, for the following values. Your code should work for any word size w ≥ 8. For reference, we show the result of evaluating the expressions for x = 0x87654321, with w = 32.

A. The least significant byte of x, with all other bits set to 0. [0x00000021]

B. All but the least significant byte of x complemented, with the least significant byte left unchanged. [0x789ABC21]

C. The least significant byte set to all ones, and all other bytes of x left unchanged. [0x876543FF]

Solution:

A.

x & 0xFF

B.

x ^ ~0x000000FF

C.

x | 0x000000FF

Practice Problem 2.13

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations and and or, it had instructions bis (bit set) and bic (bit clear). Both instructions take a data word x and a mask word m.

They generate a result z consisting of the bits of x modified according to the bits of m. With bis, the modification involves setting z to 1 at each bit position where m is 1. With bic, the modification involves setting z to 0 at each bit position where m is 1.

To see how these operations relate to the C bit-level operations, assume we have functions bis and bic implementing the bit set and bit clear operations, and that we want to use these to implement functions computing bitwise operations | and ^, without using any other C operations. Fill in the missing code below. Hint: Write C expressions for the operations bis and bic.

```c
/* Declarations of functions implementing operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions bis and bic */
int bool_or(int x, int y) {
        int result = bis(x,y);
        return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
        int result = bis(bic(x,y), bic(y,x));
        return result;
}
```

Solution:

bic(x,m) = x & ~m

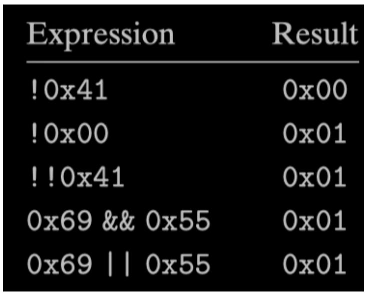
bis(x,m) = x | m

xor formula: x ^ y = (x & ~y) | (~x & y)

x ^ y = bis(bic(x,y), bic(y,x))

2.1.8 Logical Operations in C

- C also provides a set of logical operators ||, &&, and !, which correspond to the or, and, and not operations of logic.
- Differences between logical operations(|| && !) and bit-level operations (| & ~):
  1. The logical operations treat any **nonzero argument as representing true and argument 0 as representing false.**
     - View 0 as "False", while anything nonzero as "True"
     - The logical operations return either 1 or 0, indicating a result of either true or false.
     - Example:

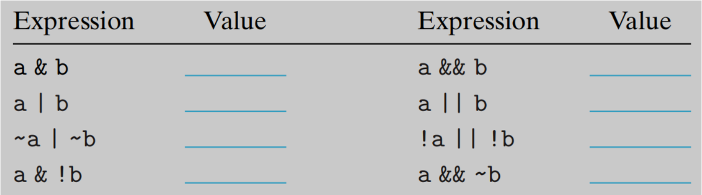


Expression	Result
!0x41	0x00
!0x00	0x01
!!0x41	0x01
0x69 && 0x55	0x01
0x69 || 0x55	0x01

  2. A second important distinction between the logical operators '&&' and '||' versus their bit-level counterparts '&' and '|' is that **the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument.**
     - Example:
       - a && 5/a will never cause a division by zero
       - p && *p will never cause the dereferencing of a null pointer.

Practice Problem 2.14

Suppose that a and b have byte values 0x55 and 0x46, respectively. Fill in the following table indicating the byte values of the different C expressions:



Expression	Value	Expression	Value
a & b		a && b	
a | b		a || b	
~a | ~b		!a || !b	
a & !b		a && ~b	

Solution:

```
a  = 0b 0101 0101
b  = 0b 0100 0110
~a = 0b 1010 1010
~b = 0b 1011 1001

a & b = 0b 0101 0101 & 0b 0100 0110 = 0b 0100 0100 = 0x44
a && b = 1

a | b = 0b 0101 0111
a || b = 1

~a | ~b = 0b 1011 1011 = 0xBB
!a || !b = 0

a & !b = 0                                      a && ~b = 1
```

Practice Problem 2.15

Using only bit-level and logical operations, write a C expression that is equivalent to x == y. In other words, it will return 1 when x and y are equal and 0 otherwise.

Solution:

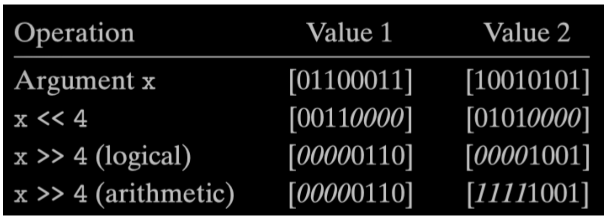
```
! (x ^ y)
```

2.1.9 Shift Operations in C (移位)

- C also provides a set of shift operations for shifting bit patterns to the left and to the right.
    - Left shift: x<<y
        - shift bit-vector x left y positions
    - Right shift: x>>y
        - shift bit-vector x right y positions
- To the left x<<k:
    - an operand x having bit representation $[x_{w-1}, x_{w-2}, \ldots, x_0]$, the C expression x << k yields a value with bit representation $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. That is, x is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k zeros.
    - Shift operations associate from left to right, so x << j << k is equivalent to (x << j) << k.
- To the right x>>k:
    - Logical - fills the left end with k zeros, giving a result Arithmetic.
        - $[0, \ldots, 0, x_{w-1}, x_{w-2}, \ldots x_k]$

  - Arithmetic - shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{w-1},\ldots,x_{w-1},x_{w-1},x_{w-2},\ldots x_k]$. This convention might seem peculiar, but as we will see, it is useful for operating on signed integer data.

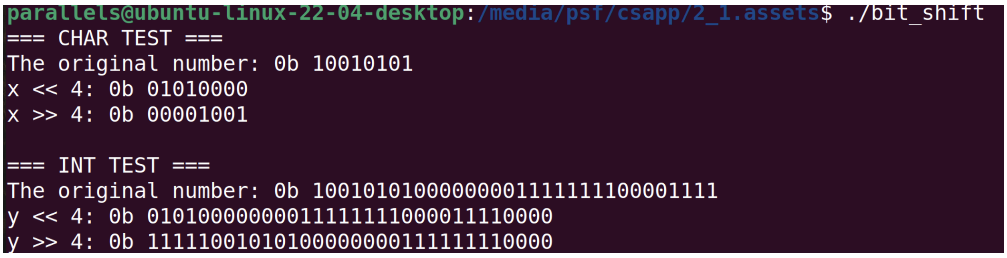
- Example:



Operation	Value 1	Value 2
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

  - Code:

```c
#include <stdio.h>

void print_CharToBin(char num)
{
        char chars_list[0x2] = "01";
        char result[0x8] = "00000000";
        int length;     // the length of result array

        if(num == 0) printf("0b 0\n");

        // Calculate the result and store into result array
        for(int i = 0;num!=0;i++)
        {
                result[i] = chars_list[num%0x2];// repeatly divide input number by 16
                                                // record the remainder into result[i]
                num /= 0x2;     // get the quotient
        }

        printf("0b ");
        for(int i = 8; i >= 0; i--)
        {
                printf("%c",result[i]);
        }
        printf("\r\n");
}

void print_UIntToBin(unsigned int num)
{
        char chars_list[0x2] = "01";
        char result[0x20] = "00000000000000000000000000000000";
        int length;     // the length of result array

        if(num == 0) printf("0b 0\n");

        // Calculate the result and store into result array
        for(int i = 0;num!=0;i++)
        {
                result[i] = chars_list[num%0x2];// repeatly divide input number by 16
                                                // record the remainder into result[i]
                num /= 0x2;     // get the quotient
        }

        printf("0b ");
        for(int i = 0x20; i >= 0; i--)
        {
                printf("%c",result[i]);
        }
        printf("\r\n");
}

int main()
{
        char x = 0b10010101;
        printf("=== CHAR TEST ===\n");
        printf("The original number: ");
        print_CharToBin(x);
        printf("x << 4: ");
        print_CharToBin(x<<4);
        printf("x >> 4: ");
        print_CharToBin(x>>4);

        printf("\n");
        printf("=== INT TEST ===\n");
        int y = 0b10010101000000000111111111100001111;
        printf("The original number: ");
        print_UIntToBin(y);
        printf("y << 4: ");
        print_UIntToBin(y<<4);
        printf("y >> 4: ");
        print_UIntToBin(y>>4);
        return 0;
}
```



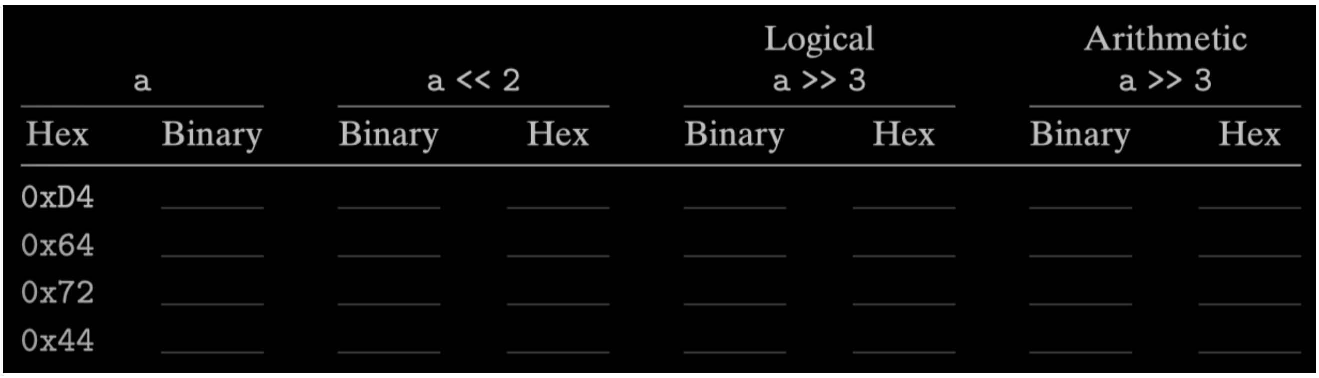
```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2_1.assets$ ./bit_shift
=== CHAR TEST ===
The original number: 0b 10010101
x << 4: 0b 01010000
x >> 4: 0b 00001001

=== INT TEST ===
The original number: 0b 10010101000000000111111111100001111
y << 4: 0b 01010000000001111111111000011110000
y >> 4: 0b 11111001010100000000011111111110000
```

- Some truths:
  - For C, The C standards do not precisely define which type of right shift should be used with signed numbers—either arithmetic or logical shifts may be used. This unfortunately means that any code assuming one form or the other will potentially encounter portability problems.
  - In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case. But, unsigned data right shifts must be logical.
  - For Java, Java has a precise definition of how right shifts should be performed.
    - The expression x >> k shifts x arithmetically by k positions, while x >>> k shifts it logically.
  - Undefined behavior: shift amount < 0 or >= word size
    - Example:

```c
char x = 0x12
x<<8 = ?
```

Practice Problem 2.16

Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.



a		a << 2		Logical a >> 3		Arithmetic a >> 3	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
0xD4							
0x64							
0x72							
0x44							

0x 87

Solution:

```
Hex:                                    0xD4
Binary:                         0b 1101 0100
a<<2 binary:  0b 0101 0000
a<<2 hex:               0x 50
Logical a>>3 binary: 0b 0001 1010
Logical a>>3 hex:        0x 1A
Arithmetic a>>3 binary: 0b 1111 1010
Arithmetic a>>3 hex:            0x FA
```

```
Hex:                                    0x 64
Binary:                         0b 0110 0100
a<<2 binary:    0b 1001 0000
a<<2 hex:               0x 90
Logical a>>3 binary: 0b 0000 1100
Logical a>>3 hex:        0x 0C
Arithmetic a>>3 binary: 0b 0000 1100
Arithmetic a>>3 hex:            0x 0C
```

```
Hex:                                    0x 72
Binary:                         0b 0111 0010
a<<2 binary:    0b 1100 1000
a<<2 hex:               0x C8
Logical a>>3 binary: 0b 0000 1110
Logical a>>3 hex:        0x 0E
Arithmetic a>>3 binary: 0b 0000 1110
Arithmetic a>>3 hex:            0x 0E
```

```
Hex:                                    0x 44
Binary:                         0b 0100 0100
a<<2 binary:    0b 0001 0000
a<<2 hex:               0x 10
Logical a>>3 binary: 0b 0000 1000
Logical a>>3 hex:    0x 08
Arithmetic a>>3 binary: 0b 0000 1000
Arithmetic a>>3 hex:      0x 08
```

```
Hex:                                    0x 87
Binary:                         0b 1000 0111
a<<2 binary:    0b 0001 1100
a<<2 hex:               0x 1C
Logical a>>3 binary: 0b 0001 0000
Logical a>>3 hex:        0x 10
Arithmetic a>>3 binary: 0b 1111 0000
Arithmetic a>>3 hex:            0x F0
```

- Addition (and subtraction) have higher precedence than shifts.
  - 1<<2 + 3<<4 -> 1 << (2+3) << 4