For floating-point number f, this function computes `(int)` f. Your function should round toward zero. If f cannot be represented as an integer (e.g., it is out of range, or it is NaN), then the function should return `0x80000000`.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

2.97 ◆◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/* Compute (float) i */
float_bits float_i2f(int i);
```

For argument `i`, this function computes the bit-level representation of `(float) i`.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

Solutions to Practice Problems

Solution to Problem 2.1 (page 73)

Understanding the relation between hexadecimal and binary formats will be important once we start looking at machine-level programs. The method for doing these conversions is in the text, but it takes a little practice to become familiar.

A. `0x25B9D2` to binary:

Hexadecimal	2	5	B	9	D	2
Binary	0010	0101	1101	1001	1101	0010

B. Binary 1100 1001 0111 1011 to hexadecimal:

Binary	1100	1001	0111	1011
Hexadecimal	C	9	7	B

C. `0xA8B3D` to binary:

Hexadecimal	A	8	B	3	D
Binary	1010	1000	1011	0011	1101

D. Binary 11 0010 0010 1101 1001 0110 to hexadecimal:

Binary	11	0010	0010	1101	1001	0110
Hexadecimal	3	2	2	D	9	6

Solution to Problem 2.2 (page 73)

This problem gives you a chance to think about powers of 2 and their hexadecimal representations.

n	2^n (decimal)	2^n (hexadecimal)
5	32	`0x20`
23	8,388,608	`0x800000`
15	32,768	`0x8000`
13	8,192	`0x2000`
12	4,096	`0x1000`
6	64	`0x40`
8	256	`0x100`

Solution to Problem 2.3 (page 74)

This problem gives you a chance to try out conversions between hexadecimal and decimal representations for some smaller numbers. For larger ones, it becomes much more convenient and reliable to use a calculator or conversion program.

Decimal	Binary	Hexadecimal
0	0000 0000	`0x00`
$158 = 16 \cdot 9 + 14$	1001 1110	`0x9E`
$76 = 16 \cdot 4 + 12$	0100 1100	`0x4C`
$145 = 16 \cdot 9 + 1$	1001 0001	`0x91`
$16 \cdot 10 + 14 = 174$	1010 1110	`0xAE`
$16 \cdot 3 + 12 = 60$	0011 1100	`0x3C`
$16 \cdot 15 + 1 = 241$	1111 0001	`0xF1`
$16 \cdot 7 + 5 = 117$	0111 0101	`0x75`
$16 \cdot 11 + 13 = 189$	1011 1101	`0xBD`
$16 \cdot 15 + 5 = 245$	1111 0101	`0xF5`

Solution to Problem 2.4 (page 75)

When you begin debugging machine-level programs, you will find many cases where some simple hexadecimal arithmetic would be useful. You can always convert numbers to decimal, perform the arithmetic, and convert them back, but being able to work directly in hexadecimal is more efficient and informative.

A. `0x605C` + `0x5` = `0x6061`. Adding `5` to hex `C` gives `1` with a carry of `1`.

B. `0x605C` − `0x20` = `0x603C`. Subtracting `2` from `5` in the second digit position requires no borrow from the third. This gives `3`.

C. `0x605C` + 32 = `0x607C`. Decimal 32 (2^5) equals hexadecimal `0x20`.

D. `0x60FA` − `0x605C` = `0x9E`. To subtract hex `C` (decimal 12) from hex `A` (decimal 10), we borrow 16 from the second digit, giving hex `F` (decimal 15). In the second digit, we now subtract 5 from hex `E` (decimal 14), giving decimal 9.

Solution to Problem 2.5 (page 84)

This problem tests your understanding of the byte representation of data and the two different byte orderings.

A.	Little endian: `78`	Big endian: `12`
B.	Little endian: `78 56`	Big endian: `12 34`

C. Little endian: `78 56 34` Big endian: `12 34 56`

Recall that `show_bytes` enumerates a series of bytes starting from the one with lowest address and working toward the one with highest address. On a little-endian machine, it will list the bytes from least significant to most. On a big-endian machine, it will list bytes from the most significant byte to the least.

Solution to Problem 2.6 (page 85)

This problem is another chance to practice hexadecimal to binary conversion. It also gets you thinking about integer and floating-point representations. We will explore these representations in more detail later in this chapter.

A. Using the notation of the example in the text, we write the two strings as follows:

```
0   0   2   7   C   8   F   8
00000000001001111100100011111000
          *********************
4    A    1    F    2     3     E     0
01001010000111111001000111110000
```

B. With the second word shifted two positions to the right relative to the first, we find a sequence with 21 matching bits.

C. We find all bits of the integer embedded in the floating-point number, except for the most significant bit having value 0. Such is the case for the example in the text as well. In addition, the floating-point number has some nonzero high-order bits that do not match those of the integer.

Solution to Problem 2.7 (page 85)

It prints `6D 6E 6F 70 71 72`. Recall also that the library routine `strlen` does not count the terminating null character, and so `show_bytes` printed only through the character ‘`r`’.

Solution to Problem 2.8 (page 87)

This problem is a drill to help you become more familiar with Boolean operations.

Operation	Result
a	[01001110]
b	[11100001]
~a	[10110001]
~b	[00011110]
a & b	[01000000]
a | b	[11101111]
a ^ b	[10101111]

Solution to Problem 2.9 (page 89)

This problem illustrates how Boolean algebra can be used to describe and reason about real-world systems. We can see that this color algebra is identical to the Boolean algebra over bit vectors of length 3.

A. Colors are complemented by complementing the values of R, G, and B. From this, we can see that white is the complement of black, yellow is the complement of blue, magenta is the complement of green, and cyan is the complement of red.

B. We perform Boolean operations based on a bit-vector representation of the colors. From this we get the following:

Blue (001)	`|`	Green (010)	=	Cyan (011)
Yellow (110)	`&`	Cyan (011)	=	Green (010)
Red (100)	`^`	Magenta (101)	=	Blue (001)

Solution to Problem 2.10 (page 90)

This procedure relies on the fact that EXCLUSIVE-OR is commutative and associative, and that a ^ $a = 0$ for any a.

Step	`*x`	`*y`
Initially	a	b
Step 1	a	a ^ b
Step 2	a ^ $(a$ ^ $b) = (a$ ^ $a)$ ^ $b = b$	a ^ b
Step 3	b	b ^ $(a$ ^ $b) = (b$ ^ $b)$ ^ $a = a$

See Problem 2.11 for a case where this function will fail.

Solution to Problem 2.11 (page 91)

This problem illustrates a subtle and interesting feature of our inplace swap routine.

A. Both `first` and `last` have value k, so we are attempting to swap the middle element with itself.

B. In this case, arguments `x` and `y` to `inplace_swap` both point to the same location. When we compute `*x ^ *y`, we get 0. We then store 0 as the middle element of the array, and the subsequent steps keep setting this element to 0. We can see that our reasoning in Problem 2.10 implicitly assumed that `x` and `y` denote different locations.

C. Simply replace the test in line 4 of `reverse_array` to be `first < last`, since there is no need to swap the middle element with itself.

Solution to Problem 2.12 (page 91)

Here are the expressions:

A. `x & 0xFF`

B. `x ^ ~0xFF`

C. `x | 0xFF`

These expressions are typical of the kind commonly found in performing low-level bit operations. The expression `~0xFF` creates a mask where the 8 least-significant bits equal 0 and the rest equal 1. Observe that such a mask will be generated regardless of the word size. By contrast, the expression `0xFFFFFF00` would only work when data type `int` is 32 bits.

Solution to Problem 2.13 (page 92)

These problems help you think about the relation between Boolean operations and typical ways that programmers apply masking operations. Here is the code:

```
/* 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;
}
```

The `bis` operation is equivalent to Boolean OR—a bit is set in `z` if either this bit is set in `x` or it is set in `m`. On the other hand, `bic(x, m)` is equivalent to `x & ~m`; we want the result to equal 1 only when the corresponding bit of `x` is 1 and of `m` is 0.

Given that, we can implement `|` with a single call to `bis`. To implement `^`, we take advantage of the property

$$x \mathbin{\hat{}} y = (x \mathbin{\&} {\sim}y) \mid ({\sim}x \mathbin{\&} y)$$

Solution to Problem 2.14 (page 93)

This problem highlights the relation between bit-level Boolean operations and logical operations in C. A common programming error is to use a bit-level operation when a logical one is intended, or vice versa.

Expression	Value	Expression	Value
`a & b`	`0x44`	`a && b`	`0x01`
`a | b`	`0x57`	`a || b`	`0x01`
`~a | ~b`	`0xBB`	`!a || !b`	`0x00`
`a & !b`	`0x00`	`a && ~b`	`0x01`

Solution to Problem 2.15 (page 93)

The expression is `!(x ^ y)`.

That is, `x^y` will be zero if and only if every bit of `x` matches the corresponding bit of `y`. We then exploit the ability of `!` to determine whether a word contains any nonzero bit.

There is no real reason to use this expression rather than simply writing `x == y`, but it demonstrates some of the nuances of bit-level and logical operations.

Solution to Problem 2.16 (page 94)

This problem is a drill to help you understand the different shift operations.

`x`		`a << 2`		Logical `a >> 3`		Arithmetic `a >> 3`	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
`0xD4`	[11010100]	[01010000]	`0x50`	[00011010]	`0x1A`	[11111010]	`0xFA`
`0x64`	[01100100]	[10010000]	`0x90`	[00001100]	`0x0C`	[11101100]	`0xEC`
`0x72`	[01110010]	[11001000]	`0xC8`	[00001110]	`0x0E`	[00001110]	`0x0E`
`0x44`	[01000100]	[00010000]	`0x10`	[00001000]	`0x08`	[11101000]	`0xE9`

Solution to Problem 2.17 (page 101)

In general, working through examples for very small word sizes is a very good way to understand computer arithmetic.

The unsigned values correspond to those in Figure 2.2. For the two's-complement values, hex digits `0` through `7` have a most significant bit of 0, yielding nonnegative values, while hex digits `8` through `F` have a most significant bit of 1, yielding a negative value.

Hexadecimal	Binary		
$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
`0xA`	[1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^2 = -6$
`0x1`	[0001]	$2^0 = 1$	$2^0 = 1$
`0xB`	[1011]	$2^3 + 2^1 + 2^0 = 11$	$-2^3 + 2^1 + 2^0 = -5$
`0x2`	[0010]	$2^1 = 2$	$2^1 = 2$
`0x7`	[0111]	$2^2 + 2^1 + 2^0 = 7$	$2^2 + 2^1 + 2^0 = 7$
`0xC`	[1100]	$2^3 + 2^2 = 12$	$-2^3 + 2^2 = -4$

Solution to Problem 2.18 (page 105)

For a 32-bit word, any value consisting of 8 hexadecimal digits beginning with one of the digits `8` through `f` represents a negative number. It is quite common to see numbers beginning with a string of `f`'s, since the leading bits of a negative number are all ones. You must look carefully, though. For example, the number `0x8048337` has only 7 digits. Filling this out with a leading zero gives `0x08048337`, a positive number.

```
4004d0:  48 81 ec e0 02 00 00    sub    $0x2e0,%rsp              A. 736
4004d7:  48 8b 44 24 a8          mov    -0x58(%rsp),%rax         B. -88
4004dc:  48 03 47 28             add    0x28(%rdi),%rax          C.  40
4004e0:  48 89 44 24 d0          mov    %rax,-0x30(%rsp)         D. -48
4004e5:  48 8b 44 24 78          mov    0x78(%rsp),%rax          E. 120
4004ea:  48 89 87 88 00 00 00    mov    %rax,0x88(%rdi)          F. 136
4004f1:  48 8b 84 24 f8 01 00    mov    0x1f8(%rsp),%rax         G. 504
4004f8:  00
4004f9:  48 03 44 24 08          add    0x8(%rsp),%rax
4004fe:  48 89 84 24 c0 00 00    mov    %rax,0xc0(%rsp)          H. 192
400505:  00
400506:  48 8b 44 d4 b8          mov    -0x48(%rsp,%rdx,8),%rax  I. -72
```

Solution to Problem 2.19 (page 107)

The functions *T2U* and *U2T* are very peculiar from a mathematical perspective. It is important to understand how they behave.

We solve this problem by reordering the rows in the solution of Problem 2.17 according to the two's-complement value and then listing the unsigned value as the result of the function application. We show the hexadecimal values to make this process more concrete.

$\vec{x}$ (hex)	x	$T2U_4(vecx)$
0xF	−1	15
0xB	−5	11
0xA	−6	10
0xC	−4	12
0x1	1	1
0x8	8	8

Solution to Problem 2.20 (page 109)

This exercise tests your understanding of Equation 2.5.

For the first four entries, the values of x are negative and $T2U_4(x) = x + 2^4$. For the remaining two entries, the values of x are nonnegative and $T2U_4(x) = x$.

Solution to Problem 2.21 (page 112)

This problem reinforces your understanding of the relation between two's-complement and unsigned representations, as well as the effects of the C promotion rules. Recall that $TMin_{32}$ is −2,147,483,648, and that when cast to unsigned it becomes 2,147,483,648. In addition, if either operand is unsigned, then the other operand will be cast to unsigned before comparing.

Expression	Type	Evaluation
`-2147483647-1 == 2147483648U`	Unsigned	1
`-2147483647-1 < 2147483647`	Signed	1
`-2147483647-1U < 2147483647`	Unsigned	0
`-2147483647-1 < -2147483647`	Signed	1
`-2147483647-1U < -2147483647`	Unsigned	1

Solution to Problem 2.22 (page 115)

This exercise provides a concrete demonstration of how sign extension preserves the numeric value of a two's-complement representation.

$$\begin{array}{llrcrcr}
\text{A.} & [1100] & -2^3+2^2 & = & -8+4 & = & -4 \\
\text{B.} & [11100] & -2^4+2^3+2^2 & = & -16+8+4 & = & -4 \\
\text{C.} & [111100] & -2^5+2^4+2^3+2^2 & = & -32+16+8+4 & = & -4
\end{array}$$

Solution to Problem 2.23 (page 116)

The expressions in these functions are common program "idioms" for extracting values from a word in which multiple bit fields have been packed. They exploit the zero-filling and sign-extending properties of the different shift operations. Note carefully the ordering of the cast and shift operations. In `fun1`, the shifts are performed on unsigned variable `word` and hence are logical. In `fun2`, shifts are performed after casting `word` to `int` and hence are arithmetic.

A.

`w`	`fun1(w)`	`fun2(w)`
`0x00000076`	`0x00000076`	`0x00000076`
`0x87654321`	`0x00000021`	`0x00000021`
`0x000000C9`	`0x000000C9`	`0xFFFFFFC9`
`0xEDCBA987`	`0x00000087`	`0xFFFFFF87`

B. Function `fun1` extracts a value from the low-order 8 bits of the argument, giving an integer ranging between 0 and 255. Function `fun2` extracts a value from the low-order 8 bits of the argument, but it also performs sign extension. The result will be a number between −128 and 127.

Solution to Problem 2.24 (page 118)

The effect of truncation is fairly intuitive for unsigned numbers, but not for two's-complement numbers. This exercise lets you explore its properties using very small word sizes.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
`1`	`1`	1	1	1	1
`3`	`3`	3	3	3	3
`5`	`5`	5	5	5	5
`C`	`4`	12	4	−4	4
`E`	`6`	14	6	−2	6

As Equation 2.9 states, the effect of this truncation on unsigned values is to simply find their residue, modulo 8. The effect of the truncation on signed values is a bit more complex. According to Equation 2.10, we first compute the modulo 8 residue of the argument. This will give values 0 through 7 for arguments 0 through 7, and also for arguments −8 through −1. Then we apply function $U2T_3$ to these residues, giving two repetitions of the sequences 0 through 3 and −4 through −1.

Solution to Problem 2.25 (page 119)

This problem is designed to demonstrate how easily bugs can arise due to the implicit casting from signed to unsigned. It seems quite natural to pass parameter `length` as an unsigned, since one would never want to use a negative length. The stopping criterion `i <= length-1` also seems quite natural. But combining these two yields an unexpected outcome!

Since parameter `length` is unsigned, the computation $0 - 1$ is performed using unsigned arithmetic, which is equivalent to modular addition. The result is then *UMax*. The $\leq$ comparison is also performed using an unsigned comparison, and since any number is less than or equal to *UMax*, the comparison always holds! Thus, the code attempts to access invalid elements of array `a`.

The code can be fixed either by declaring `length` to be an `int` or by changing the test of the `for` loop to be `i < length`.

Solution to Problem 2.26 (page 119)

This example demonstrates a subtle feature of unsigned arithmetic, and also the property that we sometimes perform unsigned arithmetic without realizing it. This can lead to very tricky bugs.

A. *For what cases will this function produce an incorrect result?* The function will incorrectly return 1 when `s` is shorter than `t`.

B. *Explain how this incorrect result comes about.* Since `strlen` is defined to yield an unsigned result, the difference and the comparison are both computed using unsigned arithmetic. When `s` is shorter than `t`, the difference `strlen(s) - strlen(t)` should be negative, but instead becomes a large, unsigned number, which is greater than 0.

C. *Show how to fix the code so that it will work reliably.* Replace the test with the following:

```
return strlen(s) > strlen(t);
```

Solution to Problem 2.27 (page 125)

This function is a direct implementation of the rules given to determine whether or not an unsigned addition overflows.

```
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y) {
    unsigned sum = x+y;
    return sum >= x;
}
```

Solution to Problem 2.28 (page 125)

This problem is a simple demonstration of arithmetic modulo 16. The easiest way to solve it is to convert the hex pattern into its unsigned decimal value. For nonzero values of x, we must have $(-^{\text{u}}_{4}\, x) + x = 16$. Then we convert the complemented value back to hex.

x		$-^{\text{u}}_{4}\, x$	
Hex	Decimal	Decimal	Hex
1	1	15	F
4	4	12	C
7	7	9	9
A	10	6	6
E	14	2	2

Solution to Problem 2.29 (page 129)

This problem is an exercise to make sure you understand two's-complement addition.

x	y	$x + y$	$x +^{\text{t}}_{5} y$	Case
−12	−15	−27	5	1
[10100]	[10001]	[100101]	[00101]	
−8	−8	−16	−16	2
[11000]	[11000]	[110000]	[10000]	
−9	8	−1	−1	2
[10111]	[01000]	[111111]	[11111]	
2	5	7	7	3
[00010]	[00101]	[000111]	[00111]	
12	4	16	−16	4
[01100]	[00100]	[010000]	[10000]	

Solution to Problem 2.30 (page 130)

This function is a direct implementation of the rules given to determine whether or not a two's-complement addition overflows.

```
/* Determine whether arguments can be added without overflow */
int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x <  0 && y <  0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum <  0;
    return !neg_over && !pos_over;
}
```

Solution to Problem 2.31 (page 130)

Your coworker could have learned, by studying Section 2.3.2, that two's-complement addition forms an abelian group, and so the expression `(x+y)-x` will evaluate to `y` regardless of whether or not the addition overflows, and that `(x+y)-y` will always evaluate to `x`.

Solution to Problem 2.32 (page 130)

This function will give correct values, except when `y` is *TMin*. In this case, we will have `-y` also equal to *TMin*, and so the call to function `tadd_ok` will indicate overflow when `x` is negative and no overflow when `x` is nonnegative. In fact, the opposite is true: `tsub_ok(x,` *TMin*`)` should yield 0 when `x` is negative and 1 when it is nonnegative.

One lesson to be learned from this exercise is that *TMin* should be included as one of the cases in any test procedure for a function.

Solution to Problem 2.33 (page 131)

This problem helps you understand two's-complement negation using a very small word size.

For $w = 4$, we have $TMin_4 = -8$. So -8 is its own additive inverse, while other values are negated by integer negation.

x		$-^{t}_{4}\, x$	
Hex	Decimal	Decimal	Hex
2	2	−2	E
3	3	−3	D
9	−9	−9	7
B	−5	5	5
C	−4	4	4

The bit patterns are the same as for unsigned negation.

Solution to Problem 2.34 (page 134)

This problem is an exercise to make sure you understand two's-complement multiplication.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	4	[100]	5	[101]	20	[010100]	4	[100]
Two's complement	−4	[100]	−3	[101]	12	[001100]	−4	[100]
Unsigned	2	[010]	7	[111]	14	[001110]	6	[110]
Two's complement	2	[010]	−1	[111]	−2	[111110]	−2	[110]
Unsigned	6	[110]	6	[110]	36	[100100]	4	[100]
Two's complement	−2	[110]	−2	[110]	4	[000100]	−4	[100]

Solution to Problem 2.35 (page 135)

It is not realistic to test this function for all possible values of `x` and `y`. Even if you could run 10 billion tests per second, it would require over 58 years to test all combinations when data type `int` is 32 bits. On the other hand, it is feasible to test your code by writing the function with data type `short` or `char` and then testing it exhaustively.

Here's a more principled approach, following the proposed set of arguments:

1. We know that $x \cdot y$ can be written as a $2w$-bit two's-complement number. Let u denote the unsigned number represented by the lower w bits, and v denote the two's-complement number represented by the upper w bits. Then, based on Equation 2.3, we can see that $x \cdot y = v2^w + u$.

   We also know that $u = T2U_w(p)$, since they are unsigned and two's-complement numbers arising from the same bit pattern, and so by Equation 2.6, we can write $u = p + p_{w-1}2^w$, where p_{w-1} is the most significant bit of p. Letting $t = v + p_{w-1}$, we have $x \cdot y = p + t2^w$.

   When $t = 0$, we have $x \cdot y = p$; the multiplication does not overflow. When $t \neq 0$, we have $x \cdot y \neq p$; the multiplication does overflow.
2. By definition of integer division, dividing p by nonzero x gives a quotient q and a remainder r such that $p = x \cdot q + r$, and $|r| < |x|$. (We use absolute values here, because the signs of x and r may differ. For example, dividing −7 by 2 gives quotient −3 and remainder −1.)
3. Suppose $q = y$. Then we have $x \cdot y = x \cdot y + r + t2^w$. From this, we can see that $r + t2^w = 0$. But $|r| < |x| \leq 2^w$, and so this identity can hold only if $t = 0$, in which case $r = 0$.

   Suppose $r = t = 0$. Then we will have $x \cdot y = x \cdot q$, implying that $y = q$.

When x equals 0, multiplication does not overflow, and so we see that our code provides a reliable way to test whether or not two's-complement multiplication causes overflow.

Solution to Problem 2.36 (page 135)

With 64 bits, we can perform the multiplication without overflowing. We then test whether casting the product to 32 bits changes the value:

```
1   /* Determine whether the arguments can be multiplied
2      without overflow */
3   int tmult_ok(int x, int y) {
4       /* Compute product without overflow */
5       int64_t pll = (int64_t) x*y;
6       /* See if casting to int preserves value */
7       return pll == (int) pll;
8   }
```

Note that the casting on the right-hand side of line 5 is critical. If we instead wrote the line as

```
int64_t pll = x*y;
```

the product would be computed as a 32-bit value (possibly overflowing) and then sign extended to 64 bits.

Solution to Problem 2.37 (page 135)

A. This change does not help at all. Even though the computation of `asize` will be accurate, the call to `malloc` will cause this value to be converted to a 32-bit unsigned number, and so the same overflow conditions will occur.

B. With `malloc` having a 32-bit unsigned number as its argument, it cannot possibly allocate a block of more than 2^{32} bytes, and so there is no point attempting to allocate or copy this much memory. Instead, the function should abort and return `NULL`, as illustrated by the following replacement to the original call to `malloc` (line 9):

```
uint64_t required_size = ele_cnt * (uint64_t) ele_size;
size_t request_size = (size_t) required_size;
if (required_size != request_size)
    /* Overflow must have occurred. Abort operation */
    return NULL;
void *result = malloc(request_size);
if (result == NULL)
    /* malloc failed */
    return NULL;
```

Solution to Problem 2.38 (page 138)

In Chapter 3, we will see many examples of the LEA instruction in action. The instruction is provided to support pointer arithmetic, but the C compiler often uses it as a way to perform multiplication by small constants.

For each value of k, we can compute two multiples: 2^k (when `b` is 0) and $2^k + 1$ (when `b` is `a`). Thus, we can compute multiples 1, 2, 3, 4, 5, 8, and 9.

Solution to Problem 2.39 (page 139)

The expression simply becomes `-(x<<`m`)`. To see this, let the word size be w so that $n = w - 1$. Form B states that we should compute `(x<<`w`) - (x<<`m`)`, but shifting `x` to the left by w will yield the value `0`.

Solution to Problem 2.40 (page 139)

This problem requires you to try out the optimizations already described and also to supply a bit of your own ingenuity.

K	Shifts	Add/Subs	Expression
7	1	1	`(x<<3) - x`
30	4	3	`(x<<4) + (x<<3) + (x<<2) + (x<<1)`
28	2	1	`(x<<5) - (x<<2)`
55	2	2	`(x<<6) - (x<<3) - x`

Observe that the fourth case uses a modified version of form B. We can view the bit pattern [11011] as having a run of 6 ones with a zero in the middle, and so we apply the rule for form B, but then we subtract the term corresponding to the middle zero bit.

Solution to Problem 2.41 (page 139)

Assuming that addition and subtraction have the same performance, the rule is to choose form A when $n = m$, either form when $n = m + 1$, and form B when $n > m + 1$.

The justification for this rule is as follows. Assume first that $m > 0$. When $n = m$, form A requires only a single shift, while form B requires two shifts and a subtraction. When $n = m + 1$, both forms require two shifts and either an addition or a subtraction. When $n > m + 1$, form B requires only two shifts and one subtraction, while form A requires $n - m + 1 > 2$ shifts and $n - m > 1$ additions. For the case of $m = 0$, we get one fewer shift for both forms A and B, and so the same rules apply for choosing between the two.

Solution to Problem 2.42 (page 143)

The only challenge here is to compute the bias without any testing or conditional operations. We use the trick that the expression `x >> 31` generates a word with all ones if `x` is negative, and all zeros otherwise. By masking off the appropriate bits, we get the desired bias value.

```
int div16(int x) {
    /* Compute bias to be either 0 (x >= 0) or 15 (x < 0) */
    int bias = (x >> 31) & 0xF;
    return (x + bias) >> 4;
}
```

Solution to Problem 2.43 (page 143)

We have found that people have difficulty with this exercise when working directly with assembly code. It becomes more clear when put in the form shown in `optarith`.

We can see that `M` is 31; `x*M` is computed as `(x<<5)-x`.

We can see that `N` is 8; a bias value of 7 is added when `y` is negative, and the right shift is by 3.

Solution to Problem 2.44 (page 144)

These "C puzzle" problems provide a clear demonstration that programmers must understand the properties of computer arithmetic:

A. `(x > 0) || (x-1 < 0)`
False. Let `x` be −2,147,483,648 ($TMin_{32}$). We will then have `x-1` equal to 2,147,483,647 ($TMax_{32}$).

B. `(x & 7) != 7 || (x<<29 < 0)`
True. If `(x & 7) != 7` evaluates to 0, then we must have bit x_2 equal to 1. When shifted left by 29, this will become the sign bit.

C. `(x * x) >= 0`
False. When `x` is 65,535 (`0xFFFF`), `x*x` is −131,071 (`0xFFFE0001`).

D. `x < 0 || -x <= 0`
True. If `x` is nonnegative, then `-x` is nonpositive.

E. `x > 0 || -x >= 0`
False. Let `x` be −2,147,483,648 ($TMin_{32}$). Then both `x` and `-x` are negative.

F. `x+y == uy+ux`
True. Two's-complement and unsigned addition have the same bit-level behavior, and they are commutative.

G. `x*~y + uy*ux == -x`
True. `~y` equals `-y-1`. `uy*ux` equals `x*y`. Thus, the left-hand side is equivalent to `x*-y-x+x*y`.

Solution to Problem 2.45 (page 147)

Understanding fractional binary representations is an important step to understanding floating-point encodings. This exercise lets you try out some simple examples.

$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$	0.11	0.75
$\frac{25}{16}$	1.1001	1.5625
$\frac{43}{16}$	10.1011	2.6875
$\frac{9}{8}$	1.001	1.125
$\frac{47}{8}$	101.111	5.875
$\frac{51}{16}$	11.0011	3.1875

One simple way to think about fractional binary representations is to represent a number as a fraction of the form $\frac{x}{2^k}$. We can write this in binary using the binary representation of x, with the binary point inserted k positions from the right. As an example, for $\frac{25}{16}$, we have $25_{10} = 11001_2$. We then put the binary point four positions from the right to get 1.1001_2.

Solution to Problem 2.46 (page 147)

In most cases, the limited precision of floating-point numbers is not a major problem, because the *relative* error of the computation is still fairly low. In this example, however, the system was sensitive to the *absolute* error.

A. We can see that $0.1 - x$ has the binary representation

$$0.000000000000000000000001100[1100]\cdots_2$$

B. Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is simply $2^{-20} \times \frac{1}{10}$, which is around 9.54×10^{-8}.

C. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$ seconds.

D. $0.343 \times 2{,}000 \approx 687$ meters.

Solution to Problem 2.47 (page 153)

Working through floating-point representations for very small word sizes helps clarify how IEEE floating point works. Note especially the transition between denormalized and normalized values.

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00	0	0	1	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	0	0.0
0 00 01	0	0	1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0.25
0 00 10	0	0	1	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{1}{2}$	0.5
0 00 11	0	0	1	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	0.75
0 01 00	1	0	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$	1	1.0
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10	1	0	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$	$\frac{3}{2}$	1.5
0 01 11	1	0	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	1.75
0 10 00	2	1	2	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$	2	2.0
0 10 01	2	1	2	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$	$\frac{5}{2}$	2.5
0 10 10	2	1	2	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$	3	3.0
0 10 11	2	1	2	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$	$\frac{7}{2}$	3.5
0 11 00	—	—	—	—	—	—	∞	—
0 11 01	—	—	—	—	—	—	*NaN*	—
0 11 10	—	—	—	—	—	—	*NaN*	—
0 11 11	—	—	—	—	—	—	*NaN*	—

Solution to Problem 2.48 (page 155)

Hexadecimal `0x359141` is equivalent to binary [1101011001000101000001]. Shifting this right 21 places gives $1.101011001000101000001_2 \times 2^{21}$. We form the fraction field by dropping the leading 1 and adding two zeros, giving

$$[10101100100010100000100]$$

The exponent is formed by adding bias 127 to 21, giving 148 (binary [10010100]). We combine this with a sign field of 0 to give a binary representation

$$[01001010010101100100010100000100]$$

We see that the matching bits in the two representations correspond to the low-order bits of the integer, up to the most significant bit equal to 1 matching the high-order 21 bits of the fraction:

```
    0    0    3    5    9    1    4    1
  00000000001101011001000101000001
              *********************
      4    A    5    6    4    5    0    4
    01001010010101100100010100000100
```

Solution to Problem 2.49 (page 156)

This exercise helps you think about what numbers cannot be represented exactly in floating point.

A. The number has binary representation 1, followed by n zeros, followed by 1, giving value $2^{n+1} + 1$.

B. When $n = 23$, the value is $2^{24} + 1 = 16{,}777{,}217$.

Solution to Problem 2.50 (page 157)

Performing rounding by hand helps reinforce the idea of round-to-even with binary numbers.

Original		Rounded	
10.111_2	$2\frac{7}{8}$	11.0	3
11.010_2	$3\frac{1}{4}$	11.0	3
11.000_2	3	11.0	3
10.110_2	$2\frac{3}{4}$	11.0	3

Solution to Problem 2.51 (page 158)

A. Looking at the nonterminating sequence for $\frac{1}{10}$, we see that the 2 bits to the right of the rounding position are 1, so a better approximation to $\frac{1}{10}$ would be obtained by incrementing x to get $x' = 0.00011001100110011001101_2$, which is larger than 0.1.

B. We can see that $x' - 0.1$ has binary representation

$$0.0000000000000000000000000[1100]$$

Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is $2^{-22} \times \frac{1}{10}$, which is around 2.38×10^{-8}.

C. $2.38 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.086$ seconds, a factor of 4 less than the error in the Patriot system.

D. $0.086 \times 2{,}000 \approx 171$ meters.

Solution to Problem 2.52 (page 158)

This problem tests a lot of concepts about floating-point representations, including the encoding of normalized and denormalized values, as well as rounding.

Format A		Format B		
Bits	Value	Bits	Value	Comments
`011 0000`	1	`0111 000`	1	
`101 1110`	$\frac{15}{2}$	`1001 111`	$\frac{15}{2}$	
`010 1001`	$\frac{25}{32}$	`0110 100`	$\frac{3}{4}$	Round down
`110 1111`	$\frac{31}{2}$	`1011 000`	16	Round up
`000 0001`	$\frac{1}{64}$	`0001 000`	$\frac{1}{64}$	Denorm → norm

Solution to Problem 2.53 (page 161)

In general, it is better to use a library macro rather than inventing your own code. This code seems to work on a variety of machines, however.

We assume that the value `1e400` overflows to infinity.

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

Solution to Problem 2.54 (page 161)

Exercises such as this one help you develop your ability to reason about floating-point operations from a programmer's perspective. Make sure you understand each of the answers.

A. `x == (int)(double) x`
Yes, since `double` has greater precision and range than `int`.

B. `x == (int)(float) x`
No. For example, when `x` is *TMax*.

C. `d == (double)(float) d`
No. For example, when `d` is `1e40`, we will get $+\infty$ on the right.

D. `f == (float)(double) f`
Yes, since `double` has greater precision and range than `float`.

E. `f == -(-f)`
Yes, since a floating-point number is negated by simply inverting its sign bit.

F. `1.0/2 == 1/2.0`
Yes, the numerators and denominators will both be converted to floating-point representations before the division is performed.

G. `d*d >= 0.0`
Yes, although it may overflow to $+\infty$.

H. `(f+d)-f == d`
No. For example, when `f` is `1.0e20` and `d` is 1.0, the expression `f+d` will be rounded to `1.0e20`, and so the expression on the left-hand side will evaluate to 0.0, while the right-hand side will be 1.0.