Week 03 Tutorial Answers

1. On a machine with 16-bit `int`s, the C expression `(30000 + 30000)` yields a negative result.

   Why the negative result? How can you make it produce the correct result?

> **Answer:**
>
> Since the two numbers in the expression are treated as `int` values, the overall expression is computed as an `int` value. In C, an `int` is a signed value, which means that one of the sixteen bits is effectively used as a sign (`+` or `-`) bit, leaving only fifteen bits to represent the magnitude. The negative result occurs because 60000 cannot be stored in fifteen bits. The maximum value is $\texttt{7FFF}_{16} = 2^{15} - 1 = 32767$.
>
> One approach to resolving the problem is to write the expression as `(30000u + 30000u)`, which will cause C to evaluate the expression as an `unsigned int` value; `unsigned int`s can use the whole sixteen bits for magnitude, allowing them to store values up to $\texttt{FFFF}_{16} = 2^{16} - 1 = 65535$.

2. Assume that the following hexadecimal values are 16-bit twos-complement. Convert each to the corresponding decimal value.

   i. `0x0013`
   ii. `0x0444`
   iii. `0x1234`
   iv. `0xffff`
   v. `0x8000`

> **Answer:**
>
> For the positive values (i.e., leftmost bit 0), convert to bits and then, for any ith bit which is 1, sum the 2^i values. For the negative values, convert them (using two-complement) to their corresponding value, and then apply the strategy for positive values.
>
> i. $\texttt{0x0013} = 0000\,0000\,0001\,0011_2 = 2^4 + 2^1 + 2^0 = 19$
> ii. $\texttt{0x0444} = 0000\,0100\,0100\,0100_2 = 2^{10} + 2^6 + 2^2 = 1092$
> iii. $\texttt{0x1234} = 0001\,0010\,0011\,0100_2 = 2^{12} + 2^9 + 2^5 + 2^4 + 2^2 = 4660$
> iv. $\texttt{0xffff} \Rightarrow -1 \times (\texttt{0x0000} + 1) = -1$
> v. $\texttt{0x8000} \Rightarrow -1 \times (\texttt{0x7fff} + 1) \rightarrow -((2^{15} - 1) + 1) = -32768$

3. Give a representation for each of the following decimal values in 16-bit twos-complement bit-strings. Show the value in binary, octal and hexadecimal.

   i. 1
   ii. 100
   iii. 1000
   iv. 10000
   v. 100000
   vi. -5
   vii. -100

> **Answer:**
>
> | | | | | |
> |---|---|---|---|---|
> | i. | 1_{10} | $0000\,0000\,0000\,0001_2$ | 01_8 | $\texttt{0x0001}_{16}$ |
> | ii. | 100_{10} | $0000\,0000\,0110\,0100_2$ | 0144_8 | $\texttt{0x0064}_{16}$ |
> | iii. | 1000_{10} | $0000\,0011\,1110\,1000_2$ | 01750_8 | $\texttt{0x03E8}_{16}$ |
> | iv. | 10000_{10} | $0010\,0111\,0001\,0000_2$ | 023420_8 | $\texttt{0x2710}_{16}$ |
> | vi. | -5_{10} | $-0000\,0000\,0000\,0101_2$ | | |
> | | | $1111\,1111\,1111\,1011_2$ | 0177774_8 | $\texttt{0xFFFB}_{16}$ |
> | vii. | -100_{10} | $-0000\,0000\,0110\,0100_2$ | | |
> | | | $1111\,1111\,1001\,1100_2$ | 0177634_8 | $\texttt{0xFF9C}_{16}$ |
>
> 100000 cannot be represented in 16 bits

4. What decimal numbers do the following single-precision IEEE 754-encoded bit-strings represent?

   a. `0 00000000 00000000000000000000000`
   b. `1 00000000 00000000000000000000000`
   c. `0 01111111 10000000000000000000000`
   d. `0 01111110 00000000000000000000000`

e. `0 01111110 11111111111111111111111`
f. `0 10000000 01100000000000000000000`
g. `0 10010100 10000000000000000000000`
h. `0 01101110 10100000101000001010000`

Each of the above is a single 32-bit bit-string, but partitioned to show the sign, exponent and fraction parts.

Answer:

All values are computed by the formula

$$\text{sign} \times (1 + \text{frac}) \times 2^{\text{exp}-127}$$

where

- **sign** is 1 if the most significant bit (m.s.b) is 0, or -1 if the m.s.b is 1
- **exp** is determined by the 8 bits following the sign bit (as a value in the range 0..255)
- **frac** is determined by the least significant 23 bits ($\text{bit}_{22} \times 2^{-1} + \text{bit}_{21} \times 2^{-2} + \cdots + \text{bit}_1 \times 2^{-22} + \text{bit}_0 \times 2^{-23}$)

a. $0.0000 = (1 + 0.0) \times 2^{0-127} = 1.0 \times 2^{-127}$ (... which is close to zero)

b. -0.0000 ... same as above, but the sign bit is 1, so -ve

c. $1.5 = (1 + 0.5) \times 2^{127-127} = 1.5 \times 2^0$

d. $0.5 = (1 + 0.0) \times 2^{126-127} = 1.0 \times 2^{-1}$

e. $0.999999 = (1 + 0.999999) \times 2^{126-127} = 1.999999 \times 2^{-1}$
(... but we're limiting this to the approximately 7 digits we can represent here)

f. $2.75 = (1 + 0.375) \times 2^{128-127} = 1.375 \times 2^1$

g. $3145728.00 = (1 + 0.5) \times 2^{148-127} = 1.5 \times 2^{21}$

h. $0.0000124165 = (1 + 0.627451) \times 2^{110-127} = 1.627451 \times 2^{-17}$
(... again, limited to approximately 7 digits)

5. Convert the following decimal numbers into IEEE 754-encoded bit-strings:

a. 2.5
b. 0.375
c. 27.0
d. 100.0

Answer:

We need to first express the number k as $(1 + \text{frac}) \times 2^n$. To work out the fraction, we divide k by the largest 2^n that is smaller than k.

a. `0 10000000 01000000000000000000000`
$= 1.25 \times 2^1 = (1 + 0.25) \times 2^{128-127}$

b. `0 01111101 10000000000000000000000`
$= 1.5 \times 2^{-2} = (1 + 0.5) \times 2^{125-127}$

c. `0 10000011 10110000000000000000000`
$= 1.6875 \times 2^4 = (1 + 0.6875) \times 2^{131-127}$ where $0.6875 = 2^{-1} + 2^{-3} + 2^{-4}$

d. `0 10000101 10010000000000000000000`
$= 1.5625 \times 2^6 = (1 + 0.5625) \times 2^{133-127}$ where $0.5625 = 2^{-1} + 2^{-4}$

6. Write a C function, `six_middle_bits`, which, given a `uint32_t`, extracts and returns the middle six bits.

Answer:

```c
uint32_t six_middle_bits(uint32_t u) {
    return (u >> 13) & 0x3F;
}
```

7. Draw diagrams to show the difference between the following two data structures:

```c
struct {
    int a;
    float b;
} x1;
union {
    int a;
    float b;
} x2;
```

If `x1` was located at `&x1 == 0x1000` and `x2` was located at `&x2 == 0x2000`, what would be the values of `&x1.a`, `&x1.b`, `&x2.a`, and `&x2.b`?

Answer:

The `struct` contains two separate components.
The `union` contains two components that occupy the same memory space.

`&x1.a==0x1000`, `&x1.b==0x1004`, `&x2.a==0x2000`, `&x2.b==0x2000`

8. How large (#bytes) is each of the following C `union`s?

    a. `union { int a; int b; } u1;`

    b. `union { unsigned short a; char b; } u2;`

    c. `union { int a; char b[12]; } u3;`

    d. `union { int a; char b[14]; } u4;`

    e. `union { unsigned int a; int b; struct { int x; int y; } c; } u5;`

    You may assume `sizeof(char)` == 1, `sizeof(short)` == 2, `sizeof(int)` == 4.

Answer:

The size of a `union` is the size of its largest variant:

a. `sizeof(u1) == 4`

b. `sizeof(u2) == 2`

c. `sizeof(u3) == 12`

d. `sizeof(u4) == 16`, with padding on string

e. `sizeof(u5) == 8`

Note that the above results may vary depending on the machine architecture and the compiler.

9. Consider the following C `union`

```
union _all {
   int   ival;
   char cval;
   char  sval[4];
   float fval;
   unsigned int uval;
};
```

If we define a variable `union _all var;` and assign the following value `var.uval = 0x00313233;`, then what will each of the following *printf*s produce:

a. `printf("%x\n", var.uval);`
b. `printf("%d\n", var.ival);`
c. `printf("%c\n", var.cval);`
d. `printf("%s\n", var.sval);`
e. `printf("%f\n", var.fval);`
f. `printf("%e\n", var.fval);`

You can assume that bytes are arranged from right-to-left in increasing address order.

Answer:

This is just a matter of interpreting the bytes/words in terms of the appropriate data type:

a. `printf("%x\n", var.uval);` gives **313233**
b. `printf("%d\n", var.ival);` gives **3224115**
c. `printf("%c\n", var.cval);` gives **3** (based on the byte ordering)
d. `printf("%s\n", var.sval);` gives **321** (based on the byte ordering)
e. `printf("%f\n", var.fval);` gives **0.000000** (actually a number very close to zero)
f. `printf("%e\n", var.fval);` gives **4.517947e-39**

The floating-point interpretation here is a *sub-normal* value, which we know because its exponent is all zero. Notably, the value of a subnormal does not have a leading one added to its mantissa, and use an exponent of 2^{-126}. We don't really cover sub-normal values, but they're excellent for understanding the floating-point interpretation rules.

Note also that `float` values are always promoted to `double` when passed to a function. See the C standard for details. (ISO 9899:2018, §6.5.2.2 ¶6)