3.70

Problem

We are given the following union declaration:

```
union ele {
  struct {
    long *p;
    long y;
  } e1;
  struct {
    long x;
    union ele *next;
  } e2;
};
```

along with the following (incomplete) function:

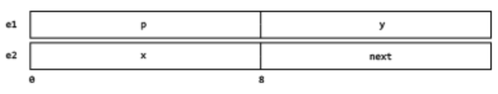
```
void proc(union ele *up) {
  up->_____ = *(_____)-____;
}
```

Questions

A. What are the offsets (in bytes) of the following fields?

```
e1.p
e1.y
e2.x
e2.next
```

Solution



Field	Offset
`e1.p`	0
`e1.y`	8
`e2.x`	0
`e2.next`	8

B. How many total bytes does the structure require?

Solution

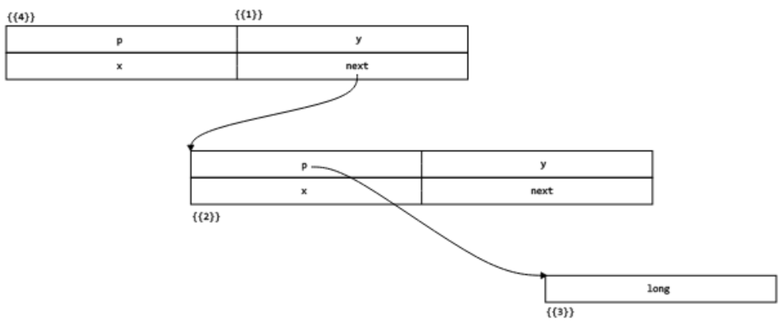
The union, `ele`, along with the structs `e1` and `e2` all take up (the same) 16 bytes in memory.

C. Given the following assembly code for `proc()`, fill in the missing expressions for `proc()`

```
void proc (union ele *up)  # up in %rdi

proc:

movq   8(%rdi), %rax

movq   (%rax),  %rdx

movq   (%rdx),  %rdx

subq   8(%rax), %rdx

movq   %rdx,    (%rdi)

ret
```

Solution

```
void proc (union ele *up)  # up in %rdi

proc:

movq   8(%rdi), %rax   # {{1}} *(up + 8) => rax  ---  either up->y or up->next

movq   (%rax),  %rdx   # {{2}} *rax => rdx  ---  *(up->next).p  (note: only getting 8 bytes not the whole union)

movq   (%rdx),  %rdx   # {{3}} *rdx => rdx  ---  *(up->next->p)

subq   8(%rax), %rdx   # rdx - *(8 + rax) => rdx

movq   %rdx,    (%rdi) # rdx => *rdi {{4}}

ret
```




```
void proc(union ele *up) {
  up->x = *(up->next->p) - (up->next->y);
}
```

2.89

Problem

On a machine where `int` has a 32-bit two's complement representation, `float` uses the 32-bit IEEE format, and `double` uses the 64-bit IEEE format, we have the following initial state:

```
int x = random();
int y = random();
int z = random();

double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression always yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

Questions

A. `(float) x == (float) dx`

False, because converting from `int` to `float` may require rounding but converting from `double` to `float` doesn't.

B. `dx - dy == (double) (x - y)`

True, since `dx` and `dy` are numerically the same as `x` and `y` respectively because `int` to `double` conversion is done exactly.

C. `(dx + dy) + dz == dx + (dy + dz)`

False, because different rounding from the addition in the parentheses may be different for each case since it rounds to even.

D. `(dx * dy) * dz == dx * (dy * dz)`

False, because different rounding from the addition in the parentheses may be different for each case since it rounds to even.

E. `dx/dx == dz/dz`

True