Floating Point (FLP)

Do not use floating-point variables as loop counters

Example

```
void funcl(void) {
    float x;

    for (x = 5.0f; x >= 1.0f; x -= 0.4f) {

        printf("%0.52f\n", x);
    }
```

```
5.0000000000000000000000000000000000000000000000000000
4.5999999046325683593750000000000000000000000000000000
4.1999998092651367187500000000000000000000000000000000
3.7999997138977050781250000000000000000000000000000000
3.3999996185302734375000000000000000000000000000000000
2.9999995231628417968750000000000000000000000000000000
2.5999994277954101562500000000000000000000000000000000
2.1999993324279785156250000000000000000000000000000000
1.7999993562698364257812500000000000000000000000000000
1.3999993801116943359375000000000000000000000000000000
```

我們平常是用十進位數，但電腦的浮點數常用二進位或十六進位運算跟儲存，一個簡單的十進位數有可能轉換成無限位的二進位或十六進位數。像圖中的 0.4 轉換完會有誤差。把十進位的 0.4 轉換成二進位的計算方式，簡單地說，我們用 2 去乘給定的十進位數，取出小數點前方的值（這是結果的一部份），再用 2 去乘剩下來的小數部份等等。依次得到 0、1、1、0，接下來的小數部份是 0.4，又回到前面出現過的內容，因此 0.4（十進位）就轉換成一個二進位的循環小數 0.0110 0110 0110 0110 0110 ……。因此，簡單的十進位 0.4 就無法完整地在計算機中儲存；換句話說，程式中寫了 0.4，但儲存到記憶體中的卻不是 0.4，而是 0.4 的近似值，於是算出來的結果就可能不夠精確了。

如圖，原本 5 - 0.4 應該是 4.6，可是因為 0.4 有誤差所以結果會差一點，變成 4.5999999 ……，而且因為這樣，迴圈就比原本預期的少做一次。

Correction version

```
5.000000
4.600000
4.200000
3.800000
3.400000
3.000000
2.600000
2.200000
1.800000
1.400000
1.000000
```

```
x = 5.0f;
float y = x;
for (size_t count = 1; count <= 11; ++count) {
    printf("%f\n", y);
    y = x - count * 0.4f;
}
```

可以用 size_t 來解決剛剛的問題，這樣就可以達到我們想要的結果，總共做了 11 次。

Ensure that floating-point conversions are within range of the new type

Example

```
void func3(void) {
    float v = -1.0f;
    unsigned int i_a = (int)v;

    printf("%u\n\n", i_a);
```

```
4294967295
```

Correction version

```
    float d;
    if (!isfinite(v)) {
        printf("isfinite(v) failed.\n");
        return;
    }
    d = trunc(v);
    if (!((d>=0.0) && (d<=(float)UINT_MAX))) {
        printf("((d>=0.0) && (d<=(float)UINT_MAX)) failed.\n")
        return;
    }
    i_a = (unsigned int)d;
    printf("%u\n", i_a);
```

```
((d>=0.0) && (d<=(float)UINT_MAX)) failed.
```

浮點數在轉換型態前要先確認它是不是在新型態的範圍內，如果沒有檢查的話出來的結果可能無法預期。Example 是-1.0 要轉換成 unsigned integer，結果變成無法預期的數，所以在轉換前要先確認好範圍，可參考 Correction version。

Preserve precision when converting integral values to floating-point type

Example

```
void func4(void) {
    long int big = 1234567891;
    float approx = big;

    printf("%d\n", FLT_DIG);  // # of decimal digits of precision

    printf("big = %ld\n", big);
    printf("approx = %f\n", approx);
    printf("big - (long int)approx = %ld\n\n", (big - (long int)approx))
```

```
6
big = 1234567891
approx = 1234567936.000000
big - (long int)approx = -45
```

Correction version

```
    assert(PRECISION(LONG_MAX) <= DBL_MANT_DIG * log2(FLT_RADIX));
    double dapprox = big;
    printf("big = %ld\n", big);
    printf("dapprox = %f\n", dapprox);
    printf("big - (long int)dapprox = %ld\n", (big - (long int)dapprox))
```

```
big = 1234567891
dapprox = 1234567891.000000
big - (long int)dapprox = 0
```

要轉換成浮點數前要先確認浮點數的精確度是否能完整表示該值，在這個 Example 裡 float 能保證的有效位數最多是 6 到 7 位，完全能保證的是 6 位，Example 中到 7 後面就亂掉了。要解決這個問題，可以用精確度較高的 double 來儲存，如 Correction version 中，就能正確儲存了。