\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Report: HW3

Author: E64061151 林友鈞 <[E64061151@mail.ncku.edu.tw](mailto:wang@xiaoming.tw)>

Class: 資訊系二乙

Description:

這份作業讓我學到如何知道各個數在記憶體中的bit pattern是什麼樣子，以前在使用這些變數都用的很理所當然，現在才知道該如何知道它在記憶體中的樣子。另外從這份作業中我也學到該如和計算IEEE 754 float的表示方式，也學到如果要表達超過int 範圍的整數，盡量用long long會比double更精準，因為double 會有精度的問題。

這份作業我是這樣做的，以下將逐一解釋函數的功能以及邏輯。

long double ComputeExponent(double n, int exp) :

這個函數是去計算n的exp次方。邏輯是這樣的，當exp>0則每次n會乘n乘exp-1次。同理，當exp < 0，則每次n會除n除以|exp|+1次。

void OutputBitPattern(long long \*x,int bits) :

這個函數會輸出bit pattern，至於為什麼用long long pointer來存，是因爲這樣32bits表示，64bits也能表示，函式只需寫一次就好。函式裡的迴圈裡的j會從最左邊的1開始逐一往右移，ex:1000...0000下一次0100...0000。此時利用&運算就能知道\*x的bit pattern是如何了。

int inputBitpattern\_int\_x32(const char \*pattern,int bits) :

這個函式需要input bit pattern，還有bits數，會output 32bits的整數。邏輯是這樣的，sum的型態是int，從pattern字串的最左邊開始scan，只要是1就去計算他的次方數，並用sum去加總。

long long inputBitpattern\_int\_x64(const char \*pattern,int bits) :

這個函式需要input bit pattern，還有bits數，會output 64bits的整數，所以回傳值是long long型態。邏輯和intputBitpattern\_int\_x32一樣，只是sum的型態變成long long。

long double inputBitpattern\_float(const char \*pattern,int bits)

這個函式需要input bit pattern，還有bits數，會output 32/64bits的浮點數，所以回傳值是long double型態。邏輯是這樣的，先scan pattern的第一個字元是否是1，如果是則此數是負數因此s = -1(s代表正負號)否則s = 1。接著如果是32bits/64bits就去計算pattern第2至第9/12個字元，計算此8bits/11bits的值是多少，此值-127/-1023就是exponent的部分。最後計算mantissa。mantissa部分第一個bits是2的-1次方，以此類推計算mantissa的值。最後計算s\*2^exponent\*(1+mantissa)就是bit pattern的值了。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Code:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

// 計算指數函數

long double ComputeExponent(double n,int exp){ // n是底數 exp是指數

int i;

double num = n;

if(exp > 0){

for(i = 1; i < exp; i++){

num \*= n;

}

}else{

for(i = 0; i <= -exp;i++)

num/=n;

}

return num;

}

// 以long long pointer 來存 位置 是因為這樣32bits 可以用 64bits也可以用

void OutputBitPattern(long long \*x,int bits){

printf("%d-bit: ",bits);

long long j = 0;

for(int i = bits-1;i >= 0; i--){

j = 1;

printf("%d",\*x&j<<i?1:0);

}

printf("\n");

}

// inputBitpattern\_int\_x32 和 inputBitpattern\_int\_x64 兩個function都是用來計算bit pattern 的整數值

// 兩者最大的不同是 sum 的型別

// 在x32 用int 這樣他就會自動overflow

// 在x64 用long long 也是同樣道理 當sing 是 1時就會自動overflow

int inputBitpattern\_int\_x32(const char \*pattern,int bits){

int sum = 0;

int i = 0;

for(int i = 0;i < strlen(pattern); i++){

if(pattern[i] == '1'){

sum += ComputeExponent(2,(bits-1-i));// bit - 1 - i 是因為最右邊的0是0次方所以要減1，這行是在算2的幾次方

}

}

// printf("integer = %d\n",sum);

return(sum);

}

long long inputBitpattern\_int\_x64(const char \*pattern,int bits){

long long sum = 0;

int i = 0;

for(int i = 0;i < strlen(pattern); i++){

if(pattern[i] == '1'){

sum += ComputeExponent(2,(bits-1-i));

}

}

// printf("integer = %.0Lf\n",sum);

return(sum);

}

// input bitpattern and output the float number

long double inputBitpattern\_float(const char \*pattern,int bits){

int s = 1;// s 指的是否是正數

int exponent = 0,i;// i 是 pattern 的 index

long double mentisa = 0;

if(pattern[0] == '1') s = -1;// 如果pattern第一個字是1帶表是負的數

int exponentbit = 3\*(bits/32) + 5; // 32bits 的exponent是8 bits, 64bits的exponentbit是11 bits

//計算exponentbit部分的值是多少

for(i = 1; i < exponentbit + 1; i++){

if(pattern[i] == '1') {

exponent += 1<<(exponentbit - i);

}

}

// printf("exponent = %d\n",exponent);

// 計算mentisa部分的值是多少

for(i = exponentbit + 1; i < bits; i++){

if(pattern[i] == '1'){

mentisa += ComputeExponent(2,exponentbit - i);// mentisa 最左邊的值是-1次方 以此類推 因此是exponentbit - i

}

}

// printf("exponent : %d\n",(exponent - (bits\*28-769)));

// printf("%LF\n",mentisa);

// printf("%Lf\n",(double)s\*ComputeExponent(2,(exponent - (bits\*28-769)))\*(1+mentisa));

// 最後的答案是 s(正負號) \* 2 ^ (exponent- 127(or 1023)) \* (1 + mentisa)

return (long double)s\*(long double)ComputeExponent(2,(exponent - (bits\*28-769)))\*(long double)(1+mentisa);

}

int main(int argc, char \*argv[])

{

if(argc < 3){

double test = 1;

printf("count is not enough\n");

return 0;

}

switch(atoi(argv[1])){

int x;

float y;

double z;

long long w;

long double r;

case 1 :

x = inputBitpattern\_int\_x32(argv[2],32);

y = inputBitpattern\_float(argv[2],32);

// printf("{\"integer\" : \"%d\",\"float\" : \"%f\"}\n",x,y);

printf("integer: %d\n",x);

printf("float: %f\n",y);

break;

case 2 :

x = atoi(argv[2]);

OutputBitPattern(&x,32);

break;

case 3:

y = atof(argv[2]);

OutputBitPattern(&y,32);

break;

case 4:

w = inputBitpattern\_int\_x64(argv[2],64);

r = inputBitpattern\_float(argv[2],64);

// printf("{\"integer\" : \"%lld\",\"float\" : \"%LF\"}\n",w,r);

printf("integer: %lld\n",w);

printf("float: %LF\n",r);

break;

case 5 :

w = atoll(argv[2]);

OutputBitPattern(&w,64);

break;

case 6 :

z = atof(argv[2]);

OutputBitPattern(&z,64);

break;

}

}

Compilation:

gcc -o hw3 hw3.c

Execution&output:

>> ./hw3 1 00111111000011001100110011001101

>> integer: 1057803469

float: 0.550000

>> ./hw3 2 1057803469

>> 32-bit: 00111111000011001100110011001101

>> ./hw3 3 0.550000

>> 32-bit: 00111111000011001100110011001101

>> ./hw3 4 1100001111011100010011111111011100101001111100110110010110100011

>> integer: -4333500818457926237

float: -8160483663085472768.000000

>> ./hw3 5 -4333500818457926237

>> 64-bit: 1100001111011100010011111111011100101001111100110110010110100011

>> ./hw3 6 -8160483663085472768.000000

>> 64-bit: 1100001111011100010011111111011100101001111100110110010110100011