**Những vấn đề với tính toán dấu chấm động**

Câu chuyện của chúng ta bắt đầu với đoạn chương trình Visual Basic (VB) sau:   
  
Dim a As Double, b As Double, s As Double   
  
a = 0.11   
  
b = 0.1   
  
s = 0.21   
  
If s = a + b Then   
  
MsgBox "True"   
  
Else   
  
MsgBox "False"   
  
End If   
  
Khi chạy, chúng ta sẽ nhận được thông báo sau:

Rõ ràng s = a+b vậy mà chương trình lại báo "False" (sai). Phải chăng đó là lỗi của Visual Basic? Bạn nào có ý nghĩ như vậy hãy thử lại với Java - một ngôn ngữ khác hẳn và hoàn toàn không phải do Microsoft cung cấp - sẽ thấy "lỗi" đó lặp lại giống hệt.   
  
Trở về với đoạn chương trình VB ban đầu, nếu đổi các biến a, b, s thành kiểu Currency thì chương trình lại "chạy đúng" như mong đợi (tức là đưa ra thông báo "True"). Dài dòng như vậy để các bạn thấy gốc rễ của vấn đề nằm ở đặc điểm của số dấu chấm động nhị phân và các phép toán liên quan tới chúng.   
  
Không ít lập trình viên chịu bó tay không lý giải được hiện tượng trên. Điều khá ngạc nhiên là các số dấu chấm động hiện diện ở khắp mọi nơi trong các hệ thống máy tính vậy mà nhiều người vẫn không biết hay không quan tâm tới chúng. Hầu hết tất cả các ngôn ngữ lập trình đều có kiểu dữ liệu dấu chấm động; các hệ thống điện toán từ máy PC đến siêu máy tính đều có bộ gia tốc dấu chấm động; hầu hết các trình biên dịch đều phải thực hiện việc dịch các thuật toán dấu chấm động một các thường xuyên và hầu như tất cả các hệ điều hành đều phải đối mặt với những trường hợp ngoại lệ của dấu chấm động (chẳng hạn bị tràn bộ nhớ).   
  
Một số dấu chấm động là cách biểu diễn cho một số trong một tập con các số hữu tỷ và thường dùng trong máy tính để biểu diễn gần đúng một số thực bất kỳ. Một cách cụ thể hơn, số dấu chấm động được thể hiện như một số nguyên hay số dấu chấm tĩnh (phần có nghĩa hay phần định trị) nhân với một cơ số (thường là cơ số 2 đối với máy tính) lũy thừa (số mũ) nguyên nào đó.   
  
a = m × be   
  
Trong một hệ thống như vậy, chúng ta chọn cơ số b và độ chính xác p (số chữ số được lưu). m (phần có nghĩa hay phần định trị) là một số có p chữ số được biểu diễn dưới dạng ±d.ddd...ddd (mỗi số là một số nguyên từ 0 đến b-1). Nếu số đầu tiên của m khác 0 thì nó được coi là đã chuẩn hóa. Một số cách mô tả có sử dụng một bit dấu riêng (s, thay thế cho -1 hoặc +1) và m buộc phải là số dương, e được gọi là số mũ.   
  
Mô hình này cho phép biểu diễn một giải số khá lớn trong một số bit nhỏ mà cách biểu diễn dấu chấm tĩnh không thể hiện được. Ví dụ, một số dấu chấm động với 4 số phần thập phân (b = 10, p = 4) và số mũ trong khoảng ±4 có thể dùng để biểu diễn các số 43210, 4.321 hay 0.0004321, nhưng sẽ không đủ chính xác để biểu diễn cho số 432.123 và 43212.3 (và do đó sẽ được làm tròn đến 432.1 và 43210). Trong thực tế số các chữ số thường lớn hơn 4.   
  
Ngoài ra, việc biểu diễn dấu chấm động thường bao gồm những giá trị đặc biệt: dương vô cực, âm vô cực và NaN (Not a Number - không phải số). Các giá trị vô cực được sử dụng khi kết quả quá lớn không thể biểu diễn được, còn NaN dùng để kết quả của những phép tính không hợp lệ hoặc kết quả không được định nghĩa.   
  
Trong ví dụ trên, các con số được biểu diễn trong hệ thập phân (b = 10), hệ thống máy tính cũng thực hiện như vậy trong hệ nhị phân (b = 2). Trong máy tính, số dấu chấm động được xác định kích thước bằng số bit dùng để lưu trữ chúng. Kích thước này là 32 bit hoặc 64 bit, thường gọi là độ chính xác đơn (single-precision) và độ chính xác kép (double-precision). Cũng có một số hệ thống đưa ra những kích thước lớn hơn như 80 bit (dòng x86) hay 128 bit (thường được thực hiện bằng phần mềm). Bạn có thể xem cách biểu diễn dấu chấm động nhị phân của các số thập phân tại trang web [http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html](https://quantrimang.com/url?q=aHR0cDovL2JhYmJhZ2UuY3MucWMuZWR1L2NvdXJzZXMvY3MzNDEvSUVFRS03NTQuaHRtbA%3D%3D).   
  
Thuật ngữ dấu chấm động xuất phát từ thực tế rằng không có số xác định các chữ số trước hoặc sau dấu chấm (dấu chấm thập phân có thể di chuyển được). Cách biểu diễn trong đó số các chữ số trước và sau dấu chấm thập phân cố định gọi là dấu chấm tĩnh. Nhìn chung, cách biểu diễn bằng dấu chấm động chậm hơn và thiếu chính xác hơn so với cách biểu diễn dấu chấm tĩnh, nhưng lại có thể biểu diễn được một vùng số lớn hơn.   
  
Một phép toán dấu chấm động là một phép toán số học được thực hiện với số dấu chấm động và thường bao hàm cả phép tính gần đúng hay phép làm tròn bởi vì kết quả của một phép toán có thể được biểu diễn một cách không chính xác. Do các phép toán với số dấu chấm động đòi hỏi nhiều năng lực điện toán nên nhiều bộ vi xử lý thường đi kèm với một chip bổ sung FPU (floating point unit) chuyên dùng cho các phép toán dấu chấm động mà chúng ta thường gọi là bộ đồng xử lý toán học.   
  
**RẮC RỐI SỐ DẤU CHẤM ĐỘNG NHỊ PHÂN**  
**1. Lỗi làm tròn**   
  
Vì số dấu chấm động nhị phân không thể biểu diễn chính xác các số thập phân nên việc sử dụng chúng sẽ không thể đảm bảo cho kết quả như khi sử dụng các phép toán thập phân. Điều này gây rất nhiều khó khăn cho việc phát triển và thử nghiệm các ứng dụng có sử dụng dữ liệu thực như thương mại hay tài chính. Dưới đây là một vài ví dụ điển hình.   
  
Lấy số 1 chia liên tục cho 10 sẽ cho kết quả như bảng dưới đây:

|  |  |
| --- | --- |
| **Thập phân** | **Nhị phân** |
| 0.1 | 0.1 |
| 0.01 | 0.01 |
| 0.001 | 9.999999E-4 |
| 0.0001 | 9.999999E-5 |
| 0.00001 | 9.999999E-6 |
| 0.000001 | 9.999999E-7 |
| 1E-7 | 9.999999E-8 |
| 1E-8 | 9.999999E-9 |
| 1E-9 | 9.999999E-10 |

Cột bên trái hiển thị các kết quả mong đợi hay kết quả nhận được khi dùng BigDecimal class của Java, cột bên phải hiển thị các kết quả nhận được bằng việc sử dụng kiểu dữ liệu float của Java. Các kết quả nhận được từ việc sử dụng kiểu dữ liệu double cũng tương tự như vậy (sẽ có nhiều số 0 hay số 9 hơn).   
  
Trong một chừng mực nào đó, những vấn đề như vậy có thể được giấu đi bằng cách làm tròn nhưng sẽ gây nhầm lẫn cho người dùng. Những lỗi này sẽ không bị phát hiện và tích lũy dần sau nhiều phép toán cho tới khi sai số quá lớn. Ví dụ dưới đây sử dụng biến kiểu double trong Java cho thấy 0.1 × 8 sẽ cho kết quả khác với việc cộng tám lần 0.1.

|  |  |
| --- | --- |
| **Chương trình** | **Kết quả** |
| public class Test{  public static void main(String args[]){  double i, j, k;  i=0.1;  j=i+i+i+i+i+i+i+i;  k=i\*8;  System.out.println(j==k?"True":"False"); System.out.println(j);  System.out.println(k);  }  } | False  0.7999999999999999 0.8 |

*Lưu ý*: Lỗi trên không thể được lặp lại trong VB cũng như Excel vì trong một số trường hợp Microsoft không hoàn toàn tuân thủ chuẩn IEEE 754. Tuy nhiên, chúng ta có thể thử và thấy Excel đôi khi cũng để lộ những kết quả "kỳ quái". Với VB, chúng ta thấy 0.5-0.4-0.1<>0 nhưng với Excel thì 0.5-0.4-0.1=0 mặc dù 0.5-0.4-0.1)\*1<>0 (cụ thể với Excel 2003, chúng tôi có kết quả 2.77556E-17).   
  
Thậm chí chỉ một phép toán đơn cũng có thể đưa ra kết quả rất không mong muốn. Chẳng hạn tính 5% thuế cho một cú gọi điện thoại công cộng có giá trị 0.70 đôla và làm tròn đến cent gần nhất. Dùng dấu chấm động nhị phân kiểu double, kết quả của 0.70 x 1.05 là 0.73499999999999998667732370449812151491641998291015625; kết quả lẽ ra phải là 0.735 (sẽ được làm tròn thành 0.74) thế nhưng con số này sau khi làm tròn lại là 0.73.   
  
Sai số cũng sẽ xuất hiện khi thực hiện phép cộng hay trừ với các số chênh lệch quá lớn. Các bạn có thể thư giãn với một vài đoạn lệnh VB dưới đây.

|  |  |
| --- | --- |
| **Chương trình** | **Kết quả** |
| Dim a As Single, b As Single, c As Single  a = 1000000  b = 0.1  c = a + b  MsgBox "c=" & c & "; c-a=" & c - a |  |
| Dim a As Single, b As Single, c As Single  a = 1000000.2  b = 1000000.1  c = a - b  MsgBox "c=" & c |  |

Ngoài ra, chúng ta nên nhớ rằng có những quy định của pháp luật hay các quy định trong nội bộ của các tổ chức mà các ứng dụng phải tuân thủ chặt chẽ về độ chính xác (trong hệ số thập phân) và phương pháp làm tròn (cho số thập phân) được dùng để tính toán. Những yêu cầu này chỉ có thể được thỏa mãn khi làm việc với cơ số 10.   
  
Qua những ví dụ trên, chúng ta học được một điều quan trọng: Hãy cẩn thận khi sử dụng các toán tử so sánh (== và !=) để so sánh kiểu dữ liệu dấu chấm động. Biểu thức:   
  
if (float\_expr1 == float\_expr2)   
  
sẽ rất ít khi được thỏa mãn vì rất có thể xảy ra các lỗi trong phép làm tròn số.   
  
**2. Thứ tự thực hiện phép toán ảnh hưởng tới kết quả**   
  
Các phép toán sử dụng hệ thống số dấu chấm động có 2 điểm quan trọng khác với tính toán trong đời sống thực. Điểm lưu ý thứ nhất là tính toán dấu chấm động không có tính kết hợp. Có nghĩa là với các số dấu chấm động nói chung thì:   
  
(x+y)+z <> x+(y+z)   
  
(x\*y)\*z <> x\*(y\*z)   
  
Tính toán dấu chấm động không có tính phân phối. Nghĩa là:   
  
x\*(y+z) <> x\*z + y\*z   
  
Tóm lại, thứ tự của phép toán có thể làm thay đổi kết quả của một phép toán dấu chấm động. Điều này rất quan trọng trong việc phân tích số hóa khi hai biểu thức toán học tương đương có thể không cho cùng một kết quả số và một biểu thức này có thể chính xác hơn biểu thức kia. Ví dụ:   
  
(1e100 - 1e100) + 1.0 có kết quả là 1.0   
  
trong khi đó   
  
(1e100 + 1.0) - 1e100 lại cho kết quả là 0.0   
  
Hai đoạn chương trình dưới đây sẽ có thể gây ra sự ngạc nhiên lớn hơn nhiều so với đoạn chương trình VB mà chúng ta đã xét ở phần mở đầu:   
  
int main(int argc, char \*\*argv) {   
  
double one = 0.1, two = 0.2, three = 0.3, six = 0.6;   
  
if((one + (two + three)) != six) {   
  
printf("0.1 + (0.2 + 0.3) != 0.6\n");   
  
}   
  
else {   
  
printf("0.1 + (0.2 + 0.3) = 0.6\n");   
  
}   
  
return 0;   
  
}   
  
Đoạn chương trình trên sẽ cho kết quả là 0.1 + (0.2 + 0.3) = 0.6   
  
Tuy nhiên, nếu chúng ta cũng tính tổng đó nhưng theo thứ tự khác:   
  
int main(int argc, char \*\*argv) {   
  
double one = 0.1, two = 0.2, three = 0.3, six = 0.6;   
  
if(((one + two) + three) != six) {   
  
printf("(0.1 + 0.2) + 0.3 != 0.6\n");   
  
}   
  
else {   
  
printf("(0.1 + 0.2) + 0.3 = 0.6\n");   
  
}   
  
return 0;   
  
}   
  
Chúng ta sẽ nhận kết quả là (0.1 + 0.2) + 0.3 != 0.6   
  
**BIỆN PHÁP KHẮC PHỤC**   
  
Vì những lý do trên, hầu hết các dữ liệu thương mại đều được lưu dưới dạng thập phân. Các phép tính trên dữ liệu thập phân đều sử dụng các phép toán thập phân, thường là thực hiện trên các số được lưu như số nguyên với lũy thừa 10. Vì vậy hầu hết các ngôn ngữ lập trình cho các ứng dụng thương mại đều hỗ trợ tính toán thập phân trực tiếp hoăc thông qua các thư viện. (Ngoại trừ một trường hợp đặc biệt và rất đáng tiếc là JavaScript/JScript).   
  
Gần như mọi ngôn ngữ lập trình đều hỗ trợ tính toán thập phân: C++, Eiffel và Python (thông qua các thư viện), Java (qua BigDecimal class), Visual Basic (với kiểu Currency mà chúng ta đã làm quen ở phần mở đầu), Visual Basic.NET và C# của Microsoft (cả hai đều cung cấp tính toán dấu chấm động thập phân sử dụng Decimal class của môi trường .NET). Trước đây các lệnh tính toán số thập phân dựng sẵn rất khó sử dụng, dễ sinh lỗi, khó bảo trì và đòi hỏi độ chính xác lớn không cần thiết khi trong một phép toán có cả hai giá trị lớn và nhỏ. Vấn đề nghiêm trọng tới mức người ta phải đưa ra Java Specification Request (JSR) 13 đề nghị cải tiến thư viện thập phân của Java và đến phiên bản Java 1.5 năm 2004 thì cải tiến đã được áp dụng.   
  
Các số thập phân thường được lưu dưới dạng BCD nên cần bộ nhớ lớn hơn khoảng 20% so với cách lưu số nhị phân. Do đó, riêng phép cộng với các số dạng BCD cũng đòi hỏi năng lực tính toán nhiều hơn 15-20% so với tính toán nhị phân thuần túy, với phép nhân các số BCD phải sử dụng thuật toán phức tạp hơn nhiều so với phép nhân số nhị phân. Tuy nhiên, nếu việc chuyển đổi sang dạng thập phân là bắt buộc thì việc tính toán trong hệ thập phân sẽ có hiệu quả hơn. Hiện tại, các phép toán với số dấu chấm động nhị phân thường được thực hiện bởi phần cứng vì hầu hết các máy tính đều hỗ trợ (bộ đồng xử lý toán học FPU mà chúng ta đều quen tên), trong khi đó tính toán với số dấu chấm động thập phân lại được thực hiện bởi phần mềm. Điều đó có nghĩa là tính toán thập phân chậm hơn so với tính toán nhị phân nhiều lần. Do vậy, số dấu chấm động nhị phân được sử dụng rộng rãi cho các công việc có khối lượng tính toán lớn, nơi mà tốc độ xử lý là mối quan tâm chủ yếu. Điều này sẽ còn tiếp diễn cho đến khi tính toán với số dấu chấm động thập phân được tích hợp vào phần cứng.   
  
Nếu sử dụng số dấu chấm động nhị phân, chúng ta nên tránh việc cộng/trừ các giá trị quá chênh lệch bằng cách thực hiện phép tính với các giá trị nhỏ nhất trước. Việc làm đó sẽ giảm bớt ảnh hưởng của phép làm tròn "không mong đợi". Sử dụng các biến kiểu double thay cho single (trong VB) hay float (trong Java) cũng giúp tránh được một số trường hợp sai lệch. Việc kiểm tra sự tương đương của các biểu thức có thể giải quyết bằng cách định nghĩa các giá trị chênh lệch gọi là "đủ nhỏ” và kiểm tra xem chênh lệch giữa 2 số/biểu thức có đủ nhỏ hay không:   
  
#define EPSILON 1.0e-7   
  
#define flt\_equals(a, b) (fabs((a)-(b)) < EPSILON)   
  
Người ta thường gọi giá trị chênh lệch này là EPSILON, ngay cả khi nó không phải là epsilon của các số dấu chấm động.   
  
Phương pháp này có thể sử dụng trong nhiều trường hợp nên khá được ưa thích. Giá trị EPSILON nói trên là một dung sai; nó là một khai báo về độ chính xác mong muốn của kết quả. Nhưng độ chính xác cần có lại phụ thuộc vào ngữ cảnh. Lấy một ví dụ minh chứng: ta có 2 số 1.25e-20 và 2.25e-20. Độ chênh lệch là 1e-20, nhỏ hơn rất nhiều so với EPSILON, nhưng rõ ràng chúng ta không cho rằng 2 số đó bằng nhau.   
  
Dưới đây là đoạn chương trình có thể cho chúng ta biết 2 số thực có bằng nhau hay không đến một số nhất định của những chữ số thập phân có nghĩa:   
  
#include <ieee754.h>   
  
int flt\_equals(float a, float b, int sigfigs)   
  
{   
  
union ieee754\_float \*pa, \*pb;   
  
unsigned int aexp, bexp;   
  
float sig\_mag;   
  
if (a == b)   
  
return 1;   
  
pa = (union ieee754\_float\*)&a;   
  
pb = (union ieee754\_float\*)&b;   
  
aexp = pa->ieee.exponent;   
  
bexp = pb->ieee.exponent;   
  
if (aexp != bexp || pa->ieee.negative != pb->ieee.negative)   
  
return 0;   
  
pa->ieee.exponent = pb->ieee.exponent = IEEE754\_FLOAT\_BIAS;   
  
sig\_mag = pow(10, -(float)sigfigs);   
  
if (fabs(a-b) < sig\_mag/2)   
  
return 1;   
  
return 0;   
  
}.