プログラミングおよび実習 I・第 13 回

2003 年 7 月 9 日

1　数値データの取り扱い

C 言語では、`float` 型や `double` 型のデータ[1]として、浮動小数点で表現された数値 (実数の近似値) を扱うことができます。今回は、このような浮動小数点データを取り扱う際に必要となる事項について解説します。

多くの C 言語の処理系では、 IEEE 標準 754 と呼ばれる規格に基づいて、浮動小数点データが取り扱われます。 IEEE 標準 754 では、数値データを表現する方式として (データの精度および表現できる数値の範囲の違う) いくつかの方式が定められていますが、ここでは、 C 言語の多くの処理系で採用されている次の 2 つの表現方法を紹介します。

単精度浮動小数点方式　単精度浮動小数点方式では、 1 つの数値データを次のような 32bit の情報として表現します。 C 言語の多くの処理系での `float` 型のデータはこの形式で表現されています。

倍精度浮動小数点方式　倍精度浮動小数点方式では、 1 つの数値データを次のような 64bit の情報として表現します。 C 言語の多くの処理系での `double` 型のデータはこの形式で表現されています。

これら 2 つの方式における s のビットは、この数値の符号を表します。 e は 8 bit または 11 bit で表現できる範囲の整数で、この数値の**指数部**と呼ばれます。また、 f は 23 bit または 52 bit で、**小数部**と呼ばれます。指数部を n bit、小数部を m bit とすると、あるビットの並びで表現される数値は次の表のように定められています。

s	e (n bit)	f (m bit)	数値
0	0	0	$+0$
1			-0
0	0	$1 \ldots 2^m-1$	$+\dfrac{f}{2^m} \times 2^{-(2^{n-1}-2)}$
1			$-\dfrac{f}{2^m} \times 2^{-(2^{n-1}-2)}$
0	$1 \ldots 2^n-2$	$1 \ldots 2^m-1$	$+\left(1+\dfrac{f}{2^m}\right) \times 2^{e-2^{n-1}+1}$
1			$-\left(1+\dfrac{f}{2^m}\right) \times 2^{e-2^{n-1}+1}$
0	2^n-1	0	$+\infty$
1			$-\infty$
	2^n-1	$1 \ldots 2^m-1$	NaN (Not a Number)[2]

[1] この他に `double` 型以上の精度を持つ `long double` 型という浮動小数点型もあります

[2] NaN (Not a Number) は、負の値の平方根など、そもそも実数として表現できない値を表すために用いられます

この表を注意深く見ると、IEEE 標準 754 の浮動小数点表現が持ついくつかの性質が分かります。数値データ扱うプログラムを作成する際には常に以下の点に注意を払う必要があります。

表現できる数値の絶対値には限界がある　IEEE 標準 754 では、単精度 (`float`) で $\pm 10^{38}$ 程度の範囲、倍精度 (`double`) で $\pm 10^{308}$ 程度の範囲の数値しか表現することはできません。計算機の扱う数値がこの範囲を越えてしまうことを**オーバーフロー**と呼びます。通常、計算結果がオーバーフローしてしまうと、その値は $-\infty$ や $+\infty$ として扱われます。次のプログラム `overflow.c` を見てみましょう。

overflow.c

```
 1 #include <stdio.h>
 2
 3 int main ()
 4 {
 5     double x, y;
 6
 7     x = 1.2345678987654321;
 8     do {
 9         printf("%.15e\n", x);
10         y = x;
11         x *= 10.0;
12     } while (x != y);
13     return 0;
14 }
```

`printf` の変換仕様 `%e` は、`double` 型の値を `1.234567e+89` のような形式で表示させるものです。ここでは `%.15e` としていますので、小数点以下 15 桁まで表示されます。このプログラム `overflow.c` を実行すると次のようになります。

overflow.c の実行例

```
s1609h017% ./overflow
1.234567898765432e+00
1.234567898765432e+01
1.234567898765432e+02
1.234567898765432e+03
1.234567898765432e+04

    ⋮

1.234567898765432e+300
1.234567898765432e+301
1.234567898765432e+302
1.234567898765432e+303
1.234567898765432e+304
1.234567898765432e+305
1.234567898765432e+306
1.234567898765432e+307
1.234567898765432e+308
Infinity
s1609h017%
```

表現された数値には誤差がある　浮動小数点方式で表現できるのは有限個の値でしかありませんから、与えられた実数値がそのまま表現できない場合、表現可能な数値の中から真の値にもっとも近い値を選

んで表現します。このため、単精度の場合で最大 $\frac{1}{10^6}$ 程度、倍精度の場合で最大 $\frac{1}{10^{15}}$ 程度の誤差を含む近似値として数値が表現されます。つまり、浮動小数点データの精度は単精度で有効数字 6 桁程度、倍精度で有効数字 15 桁程度ということになります。次のプログラムを見てみましょう。

rounderr.c

```
 1 #include <stdio.h>
 2
 3 int main ()
 4 {
 5     int i;
 6     double x = 3.456789;
 7     double y = 9.876543;
 8     double z = x/y*y;
 9
10     printf("x = %.18e\n", x);
11     printf("y = %.18e\n", y);
12     printf("z = %.18e\n", z);
13
14     for (i = 0; i < 1000000; i++)
15         x += 1.0e-18;
16
17     printf("x + 1.0e-12 = %.18e\n", x);
18     return 0;
19 }
```

このプログラム rounderr.c を実行すると次のようになります。x/y*y の計算結果である z が x とは等しくなっていないことに注意しましょう。

rounderr.c の実行例

```
s1609h017% ./rounderr
x = 3.456789000000000112e+00
y = 9.876542999999999850e+00
z = 3.456788999999999668e+00
x + 1.0e-12 = 3.456789000000000112e+00
s1609h017%
```

また、プログラム後半の for 文の繰り返しでは、x に 10^{-18} を 10^6 回足し込んでいますので、本来なら最後の printf が実行される時点の x の値は 3.456789000001 になるはずです。ところが、実際には x の値は全く変化していません。これは、1 回ごとに足し込まれる 10^{-18} という値が x の絶対値に対して誤差以下の絶対値しか持っていないので、加算を行っても x の値が変化しないからです。このように、全体の数値にとっては誤差程度の絶対値しか持たない数値を何度も加えたり引いたりする場合には、大きな誤差が生じる[2]可能性がありますので注意が必要です。

絶対値が非常に小さいと誤差はさらに大きくなる　数値の絶対値が、単精度 (float) で 10^{-38} 以下、倍精度 (double) で 10^{-308} 以下の場合、その数値が 0 に近づくにつれて、浮動小数点表現の小数部は上位のビットからだんだんと 0 になっていき、やがてすべてのビットが 0 となります。これに従って近似値としての精度も落ちていくことになります。次のプログラムは rounderr.c の前半部分で使っていた値を単に 10^{320} 分の 1 にしただけのプログラムですが、相対的な誤差が非常に大きくなっていることが分かります。

[2] この現象を**情報落ち**と呼びます

極めて近い 2 つの数値の間で減算を行った場合も、計算結果の絶対値が非常に小さくなって数値の精度が悪くなることがありますので注意が必要です[3]。

rounderr0.c

```
 1 #include <stdio.h>
 2
 3 int main ()
 4 {
 5     double x = 3.456789e-320;
 6     double y = 9.876543e-320;
 7     double z = x/y*y;
 8
 9     printf("x = %.18e\n", x);
10     printf("y = %.18e\n", y);
11     printf("z = %.18e\n", z);
12     return 0;
13 }
```

rounderr0.c の実行例

```
s1609h017% ./rounderr0
x = 3.4569773239512020 70e-320
y = 9.8763722603665184 18e-320
z = 3.4569773239512020 70e-320
s1609h017%
```

rounderr.c では一致していなかった z と x の値が一致するようになったのは、このくらい絶対値が小さくなると、double 型で表現できる数値の間隔が拡がってしまって、2 つ数値がたまたま同じ表現になってしまったためです。

次のプログラム underflow.c の実行例では、絶対値が 10^{-308} を切った辺りから誤差が大きくなっていき、ついには 0.0 になってしまっています。計算機の扱う数値の絶対値が小さくなりすぎて、一定の精度が保てなくなってしまうことを**アンダーフロー**と呼びます。計算結果がアンダーフローして、0 と区別がつかなくなってしまった場合、通常、その値は -0.0 あるいは $+0.0$ として扱われます。この -0.0 と $+0.0$ の 2 つはどちらも 0 という数値を表していますが、アンダーフローが正負どちらの側から起こったのかが区別できるようになっています[4]。

underflow.c

```
 1 #include <stdio.h>
 2
 3 int main ()
 4 {
 5     double x, y;
 6
 7     x = 1.2345678987654321;
 8     do {
 9         printf("%.15e\n", x);
10         y = x;
11         x /= 10.0;
12     } while (x != y);
```

[3] この現象を**桁落ち**と呼びます

[4] たとえば、C の等値演算子 == で比較すると、-0.0 == +0.0 は真 (1) となりますが、printf で %f を変換仕様として、その値を出力すると、それぞれ -0.000000 と 0.000000 のように異なる表示がされることがあります

```
13      return 0;
14 }
```

underflow.c の実行例

```
s1609h017% ./underflow
1.234567898765432e+00
1.234567898765432e-01
1.234567898765432e-02
1.234567898765432e-03
1.234567898765432e-04

    ⋮

1.234567898765432e-305
1.234567898765432e-306
1.234567898765432e-307
1.234567898765432e-308
1.234567898765431e-309
1.234567898765456e-310
1.234567898765456e-311
1.234567898765456e-312
1.234567898760515e-313
1.234567898908735e-314
1.234567900884997e-315
1.234567876181715e-316
1.234567777368586e-317
1.234566295171648e-318
1.234571235828107e-319
1.234670048957275e-320
1.235164114603116e-321
1.235164114603116e-322
9.881312916824931e-324
0.000000000000000e+00
s1609h017%
```

10 進表記での有限小数でも誤差は存在する　IEEE 標準 754 では、たとえば、10 進表記では有限小数となる 0.1 という値も、2 進表記では循環小数となりますので、浮動小数点データとしてこれを正確に表現することはできません。このため、次のプログラム steps.c の実行例で分かるように、0.1 を 10 回 0.0 に足し込んでも 1.0 と等しくなるとは限りませんので注意が必要です。一方、絶対値のあまり大きくない整数は誤差なしに表現することが可能です[5]。

steps.c

```
 1 #include <stdio.h>
 2
 3 int main ()
 4 {
 5     double x = 0.0;
 6     int i;
 7
 8     for (i = 0; i < 10; i++) {
 9         x += 0.1;
10         printf("%.18e\n", x);
11     }
```

[5]おおよそ、単精度で 6 桁まで、倍精度で 15 桁までの整数は誤差なしに表現可能です

```
12      return 0;
13 }
```

steps.c の実行例

```
s1609h017% ./steps
1.00000000000000056e-01
2.00000000000000111e-01
3.00000000000000444e-01
4.00000000000000222e-01
5.00000000000000000e-01
5.99999999999999778e-01
6.99999999999999556e-01
7.99999999999999334e-01
8.99999999999999112e-01
9.99999999999998890e-01
s1609h017%
```

この steps.c の for 文を、もし

```
for (x = 0.1; x != 10.0; x += 0.1)
    printf("%.18e\n", x);
```

のように書いてしまうと、このプログラムは永遠に x の値を表示し続けることでしょう。

C の等値演算子 == や不等演算子 != で float 型や double 型のデータを比較することには、ほとんど意味がないことに注意してプログラムを書いてください。2 つの数値 x と y がほぼ等しいかどうかを調べる時には、

fabs(x - y) < 1.0e-10

にような条件式で判定することを考えましょう[6]。

2 収束計算

前節で解説した点に注意して、与えられた実数 x に対する正弦関数の値 $\sin x$ を計算するプログラムを書いてみます[7]。次のプログラム showsin.c は、 0.0 から始めて 0.2 に至るまで、 0.01 間隔で正弦関数の値を 21 回表示するものです。

showsin.c

```
 1 #include <stdio.h>
 2
 3 #define ERR        (1.0e-15)
 4
 5 double mysin(double x)
 6 {
 7     double s, d;
 8     int n;
 9
10     n = 1;
```

[6]関数 fabs は extern double fbas(double); と宣言できる C の数学ライブラリ関数の 1 つで、引数として与えられた数値の絶対値を返します

[7]C では、extern double sin(double); と宣言できる数学ライブラリ関数 sin を使って正弦関数の値を計算することができますが、ここでは自前のプログラムで計算することにします

```
11      s = d = x;
12      do {
13          d *= -x/++n;
14          d *= x/++n;
15          s += d;
16      } while (d < -ERR || ERR < d);
17      return s;
18  }
19
20  int main()
21  {
22      int i;
23      double x;
24
25      for (i = 0; i <= 20; i++) {
26          x = i*0.01;
27          printf ("sin(%.2f) = %.15f\n", x, mysin(x));
28      }
29      return 0;
30  }
```

正弦関数の値の計算は、関数 mysin の中で次のようなマクローリン展開を利用した収束計算で行っており、$\left|\frac{x^n}{n!}\right|$ の値が ERR (10^{-15}) 以下になった時に計算を打ち切っています。

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots$$

この showsin.c を実行すると次のようになります。

showsin.c の実行例

```
s1609h017% ./showsin
sin(0.00) = 0.000000000000000
sin(0.01) = 0.009999833334167
sin(0.02) = 0.019998666693333
sin(0.03) = 0.029995500202496
sin(0.04) = 0.039989334186634
sin(0.05) = 0.049979169270678
sin(0.06) = 0.059964006479445
sin(0.07) = 0.069942847337533
sin(0.08) = 0.079914693969173
sin(0.09) = 0.089878549198011
sin(0.10) = 0.099833416646828
sin(0.11) = 0.109778300837175
sin(0.12) = 0.119712207288919
sin(0.13) = 0.129634142619695
sin(0.14) = 0.139543114644236
sin(0.15) = 0.149438132473599
sin(0.16) = 0.159318206614246
sin(0.17) = 0.169182349066996
sin(0.18) = 0.179029573425824
sin(0.19) = 0.188858894976501
sin(0.20) = 0.198669330795061
s1609h017%
```

関数 main の中の for 文に注意してください。この部分は、つい

```
for (x = 0.0; x <= 2.0; x += 0.01) {
    printf ("sin(%.2f) = %.15f\n", x, mysin(x));
}
```

のようなプログラムにしたくなりますが、これでは x の値の誤差により、$x = 2.0$ のときの表示 (最終行) がされないことがあります。これは、`double` 型のデータの計算で、0.01 を 0.0 に 20 回足し込むと、誤差によって 2.0 をわずかに越えてしまうことがあるためです。

プログラミングおよび実習 I・第 13 回・終り