Introdução à Programação

1.Ano LCC-MIERSI
DCC - FCUP

Nelma Moreira

Aula 6

Números em vírgula flutuante

Representação de um número em vírgula flutuante tem três componentes:

* um sinal, s
* uma fracção, f
* um expoente, e
* numa base, b

Ambos com um número fixo de dígitos (p.e, 4 e 1).

(-1)^s *f * b^q

Norma IEEE 754 - Vírgula Flutuante

Name	Common name	Base	Digits	E min	E max	Notes	Decimal digits	Decimal E max
binary16	Half precision	2	10+1	-14	+15	storage, not basic	3.31	4.51
binary32	Single precision	2	23+1	-126	+127		7.22	38.23
binary64	Double precision	2	52+1	-1022	+1023		15.95	307.95
binary128	Quadruple precision	2	112+1	-16382	+16383		34.02	4931.77
decimal32		10	7	-95	+96	storage, not basic	7	96
decimal64		10	16	-383	+384		16	384
decimal128		10	34	-6143	+6144		34	6144

- Embora as representações usadas nos computadores usem a base 2, aqui vamos considerar a base 10, dado que os problemas levantados são análogos.
- Se a base for 10:
  - +2.000×100 ou +2.000E + 0
  - +3.000×10-1 ou +3.000E - 1

Notação	Número
+1.000E+0	1
+3.300E+5	33000
-8.22E-03	-0.008223
+0.000E+0	0

Aritmética

Muito menos precisa e mais complicada do que com inteiros...

Exemplo: Adicionar +2.000E + 0 e 3.000E - 1 (2.0 e 0.3)

* Acrescentar um dígito a cada um para minimizar erros: +2.0000E + 0 e 3.0000E - 1
* Colocar ambos os números o mesmo expoente (do maior): +2.0000E + 0 e 0.3000E + 0
* Adicionar as fracções: +2.3000E + 0 (2.3)

* Normalizar: alterar o expoente de modo a que à esquerda do ponto só esteja um dígito não nulo. Já está neste caso
* Se o dígito que se acrescentou for maior do que 5, arredondar o dígito seguinte; senão truncar:

+2.300E + 0 (2.3)

Erros de arredondamento

Exemplo: 1/3 + 1/3 não é 2/3

* 1/3 é 3.333E - 1
* 2/3 é 6.667E - 1
* e 3.333E - 1 + 3.333E - 1 é 6.666E - 1

Exemplo: 1 - 1/3 -1/3 - 1/3 não é 0

* 1.000E + 0,
* -0.333E + 0
* -0.333E + 0,
* -0.333E + 0

igual a 0.0010E + 0, isto é, 1.000E - 3

Vírgula flutuante versus inteiros

Tipos das variáveis do C: `float` ou `double`

```
main()
{int i;
float x;
x = 1.0/2.0;
i = 1/2;
printf("x=%f i=%d
\n",x,i);
x=1/2 +1/2;
printf("x=%f\n",x);
x=3.0/2.0;
i=x;
printf("x=%f x=%e i=%d
\n",x,x,i);
}
```

Execução:

```
$ gcc fi.c -o fi
$ fi
x=0.500000 i=0
x=0.000000
x=1.500000 x=1.500000e+00
i=1
```

Conversões

* **Operações aritméticas:** se os operandos são de tipo `int`, o resultado é `int`; se pelo menos um dos operandos não é inteiro, o resultado é em vírgula flutuante.

* `float => int`: a parte fracionária é truncada

* `int => float`: se o inteiro for representável, pode não o ser exactamente ...

* O problema complica-se se se tratar de inteiros sem sinal(`unsigned int`)

* E também há conversões: `double => float` e `float => double`, etc

* Para se obter um cálculo em vírgula flutuante com valores inteiros podemos usar um:
* Operador unário de conversão (cast): `(float)`, `(double)`

```
media = (float) soma / n;
```

  * que cria uma cópia da variável em vírgula flutante e usa-a na operação.
  * Neste caso, os restantes operandos são promovidos a `float`
  * Claro que se `media` for tipo `int` o valor final é truncado.

Cálculo aproximado do valor de e

Sendo e o número de Nepar, tem-se que e = 1 + 1 + 1/2! +...+ 1/n! +... calcular a soma da sucessão até ao termo de ordem n.

Programa em C

```
#include<stdio.h>

main()
{
   int n, i, f = 1;
   float s = 1.0;
   scanf("%d",&n);
   for(i = 1; i <= n; i++)
{
      f = f * i;
      s = s+(1/(float) f);
   }
   printf("valor aprox. de
e=%10.8f\n",s);
}
```

Execução:

```
$ exp
5
valor aprox. de
e=2.71666694
$ exp
10
valor aprox. de
e=2.71828198
$ exp 30
30
valor aprox. de
e=2.71828198
$ exp 40
40
valor aprox. de e=
Inf+
```

Regras de Promoção

* Especificam como os tipos podem ser convertidos noutros tipos sem perder informação.
* Aplicam-se em expressões que envolvem valores de tipos diferentes.
* Para cada valor é automaticamente criada uma cópia cujo tipo é o do tipo mais alto da expressão, segundo a seguinte hiearquia de tipos:

Tipo	Conversão no `printf`
long double	`%Lf`
double	`%g`
float	`%f`
unsigned long int	`%lu`
long int	`%ld`
unsigned int	`%u`
int	`%d`
short	`%hd`
char	`%c`

Coersão de argumentos

* Os protótipos das funções forçam que os argumentos com que são chamadas sejam do tipo dos respectivos parâmetros.
* Se os argumentos não forem do tipo adequado, estes são convertidos antes da função ser chamada.

* Vimos que na biblioteca de funções matemáticas, os parâmetros eram sempre `double`.
* Assim, em `sqrt(2)` o `2` é promovido a `double` antes da função ser chamada.
* Mas é necessário incluir o ficheiro de cabeçalho `math.h`

Intervalo

5 minutos

Geração de números pseudo-aleatórios

* Função `rand()`

```
i = rand();
```

gera um inteiro entre `0` e `RAND_MAX`, com igual probabilidade de ocorrer.

O valor da constante simbólica `RAND_MAX` e o protótipo da função encontram-se em `<stdlib.h>`.

```c
#include <stdlib.h>
#include <stdio.h>
main() {
   int i;
   for(i = 1; i <= 100; i++)printf("%d\n",rand());
}
```

Usualmente pretendem-se valores noutros intervalos... Se for entre `0` e `a-1`:

```
srand() % a
```

Simulação do lançamento duma moeda: cara 0 e coroa 1

```
#include <stdlib.h>
#include <stdio.h>
main() {
   int i;
   for(i=1; i<=10; i++) {
     if (rand() % 2)
       printf("Cara\n");
     else
       printf("Coroa \n");
   }
}
```

Execução:

```
% moeda
Cara
Coroa
Cara
Cara
Cara
Cara
Coroa
Coroa
Cara
Cara
```

Alice e Bob atiram a moeda ao ar

Lançando uma moeda ao ar repetidamente, a Alice ganha o jogo se aparecer primeiro uma determinada sequência de três valores consecutivos e o Bob ganha se aparecer primeiro outra dessas sequências.

1. A Alice escolhe a sequência 000 (cara, cara, cara) e o Bob a sequência 111
2. A Alice escolhe a sequência 001 (cara, cara, cara) e o Bob a sequência 011

* Quem tem mais probabilidade de ganhar? Isto é, em vários jogos quem ganha mais vezes? Para cada uma das escolhas, serão as sequências da Alice e do Bob equiprováveis?

* Vamos simular N lançamentos e para cada jogador contabilizar os jogos ganhos e qual a sua frequência (número de jogos ganhos/ número de jogos total).

Algoritmo para um jogo

Tem de haver pelo menos 3 lançamentos...

```
ant2 = rand()% 2
ant = rand()% 2
Enquanto (1) fazer
    t = rand() % 2
    se saiu a sequencia da Alice(p.e ant2==0 && ant==0
&& t==1)
        entao Escreve ``Alice ganha'' e parar
    se saiu a sequencia do Bob (p.e ant2==0 && ant==1
&& t==1)
        entao Escreve ``Bob ganha'' e parar
    ant2 = ant
    ant = t
```

Programa em C para MAXJ jogos

```
#include <stdlib.h>
#include <stdio.h>
#define MAXJ 1000
#define AL1 0
#define AL2 0
#define AL3 1
#define BOB1 0
#define BOB2 1
#define BOB3 1

main() {
  int i,a=0,b=0,t,
      ant, ant2;
printf("Alice Bob\n");
printf("----------\n");
for(i=1; i<MAXJ;i++) {
  ant2 = rand()%2;
  ant = rand()%2;
  while (1) {
    t = rand()%2;
    if (ant2 == AL1 &
        ant == AL2 &
        t == AL3)
      {a++; break;}
    else
    if (ant2 == BOB1 &
      ant == BOB2 &
      t == BOB3) {b++;
                  break;}
    else {
      ant2 = ant;
      ant = t;}
    }
}
printf("%2d %2.1f %2d
%2.1f\n",a,
((float) a / MAXJ), b,
((float) b / MAXJ));
}
```

Execução

Alice ganha se aparecer primeiro 000,
Bob se aparecer primeiro 111

```
% alice
Alice   Bob
----------------
516 0.5 483 0.5
```

Conclusão:

Como era de esperar, têm a mesma probabilidade de ganhar!

Execução

Alice ganha se aparecer primeiro 001,
Bob se aparecer primeiro 011

```
% alice
Alice   Bob
---------------
655 0.7 344 0.3
```

Conclusão:

Alice tem maior probabilidade de ganhar!

Porquê? A prova analítica é complicada mas o método probabilístico (de Monte Carlo) permite uma aproximação...

Sementes da geração de pseudo-aleatórios

Se executarmos várias vezes o programa anterior, a sequência de valores gerada é sempre a mesma! Não é muito aleatório!

```
% alice1
Alice   Bob
----------------
655 0.7 344 0.3
% alice1
Alice   Bob
----------------
655 0.7 344 0.3
% alice1
Alice   Bob
----------------
655 0.7 344 0.3
```

Função `srand()`

* A função `rand()` gera uma sequência de valores que se repete sempre que o programa é executado.
* A semente da sequência é sempre a mesma (1)!
* Para produzir uma sequência diferente é necessário, mudar a semente usando a função `srand()`, cujo argumento inteiro (sem sinal) é a nova semente e que não retorna nenhum valor.

```
srand(41);
```

* Se se pretender uma sequência diferente e o utilizador não seja obrigado a introduzir a semente, podemos usar uma função que retorna o valor do relógio do computador em segundos (e cujo protótipo está em `time.h`):

```
srand(time(NULL));
```

Números aleatórios num intervalo

* Para gerar inteiros entre a e a+b-1:

```
i=a + rand() % b;
```

* Simulação de lançamentos de um dado
  * Valores entre 1 e 6:

```
1 + rand() % 6
```

Determinar a frequência absoluta de cada número de 1 a 6, em 6000 lançamentos

Algoritmo

* Definir um contador para cada um dos valores: `f1, f2, f3, f4, f5` e `f6`.
* Para cada valor gerado (entre 1 e 6), incrementar o contadocorrespondente.

```
main() {
   int vez,face,f1 = 0,f2 = 0,f3 = 0,f4 = 0,f5 = 0,f6 = 0;
   for(vez = 1; vez <= 6000; vez++) {
      face = 1 + (rand() % 6);
      if (face == 1) ++f1;
      else if (face == 2) ++f2;
      else if (face == 3) ++f3;
      else if (face == 4) ++f4;
       else if (face == 5) ++f5;  else ++f6;
     }
   }
   printf("1   \t2    \t3    \t4    \t5    \t6    \n\n");
   printf ("%4d\t%4d\t%4d\t%4d\t%4d\t%4d\t
                  \n",f1,f2,f3,f4,f5,f6);
}
```

É preferível usar uma nova instrução...

```
main() {
  int vez, face, f1 = 0,f2 = 0,f3 = 0,f4 = 0,f5 = 0,f6 =0;
  for(vez=1; vez<=6000; vez++) {
    face = 1 + (rand() % 6);
    switch(face) {
    case 1: ++f1; break;
    case 2: ++f2; break;
    case 3: ++f3; break;
    case 4: ++f4; break;
    case 5: ++f5; break;
    default: ++f6
    }
  }

  printf("1  \t2   \t3   \t4   \t5   \t6   \n\n");

  printf ("%4d\t%4d\t%4d\t%4d\t%4d\t%4d\t
           \n",f1,f2,f3,f4,f5,f6);

}
```

Execução

* Cada um devia ocorrer 1000 vezes...

```
% cc      dados1.c    -o dados1
% dados1
```

```
1          2          3          4          5          6

980        993        1030       1009       1002       986

%
```

Instrução de escolha `switch`

Permite a selecção de uma de várias alternativas.

```
switch(Expr){
        case Exp1: Insts1
        case Exp2: Insts2
        ...
        default: Instsd
}
```

A `Expr` é calculada. Se for igual a `Exp1`, a instrução `Insts1` é executada,..., etc. Se não for igual a nenhuma, é executada `Instsd`.

* As expressões `Exp1`, `Exp2`,...têm de ser constantes.

* Cada grupo de instruções `Insts1`, `Insts2` é normalmente terminado com a instrução `break`. Se não for, a execução continua com as instruções à frente (nas outras alternativas).

* A parte de `default` é opcional.

switch (expr) {case exp1:inst1... case expn: instn}