13 Números Reais - Tipo float

Ronaldo F. Hashimoto e Carlos H. Morimoto

Até o momento nos limitamos ao uso do tipo inteiro para variáveis e expressões aritméticas. Vamos introduzir agora o tipo real.

Ao final dessa aula você deverá saber:

- Declarar, ler e imprimir variáveis do tipo `float`.
- Calcular o valor de expressões aritméticas envolvendo reais.
- Utilizar variáveis reais em programas.

Para entender como são representados variáveis do tipo real, vamos falar um pouco sobre como os números inteiros e reais são representados no computador.

13.1 Representação de números inteiros

Os circuitos eletrônicos do computador armazenam a informação na forma binária (também chamada de digital). Um dígito binário pode assumir apenas 2 valores, representados pelos símbolos 0 (zero) e 1 (um), e que nos circuitos podem corresponder, por exemplo, a uma chave aberta/fechada, a um capacitor carregado/descarregado, etc. Esse elemento básico é conhecido como **bit**.

Os bits (dígitos binários) podem ser combinados para representar números da mesma forma que os dígitos decimais (dígitos de zero a nove), através de uma notação posicional, ou seja, o número 12 na base decimal equivale ao resultado da expressão $\boxed{1} \times 10^1 + \boxed{2} \times 10^0$. Essa mesma quantia pode ser representada por 1100 na base binária pois equivale ao resultado da expressão $\boxed{1} \times 2^3 + \boxed{1} \times 2^2 + \boxed{0} \times 2^1 + \boxed{0} \times 2^0$.

Por razões históricas, a memória do computador é dividida em bytes (conjunto de 8 bits), por isso a memória do seu computador pode ter, por exemplo, 128MB (mega bytes, onde 1MB $= 2^{20}$ bytes) e o disco rígido 40GB (giga bytes, onde 1GB $= 2^{30}$ bytes). Com um byte, é possível representar $2^8 = 256$ valores (todas as possíveis configurações de 8 bits de 00000000 a 1111111). Então os números decimais inteiros com 8 bits possíveis estão compreendidos de

$$\boxed{0} \times 2^7 + \boxed{0} \times 2^6 + \boxed{0} \times 2^5 + \boxed{0} \times 2^4 + \boxed{0} \times 2^3 + \boxed{0} \times 2^2 + \boxed{0} \times 2^1 + \boxed{0} \times 2^0 = 0$$

a

$$\boxed{1} \times 2^7 + \boxed{1} \times 2^6 + \boxed{1} \times 2^5 + \boxed{0} \times 2^4 + \boxed{1} \times 2^3 + \boxed{1} \times 2^2 + \boxed{1} \times 2^1 + \boxed{1} \times 2^0 = 255$$

ou seja, com 8 bits podemos representar inteiros de 0 a 255 (256 valores).

Nos computadores digitais, para representar de números negativos, é comum usar um bit para sinal. Se o bit de sinal é 0 (zero), então o número é positivo; caso contrário, o número é negativo. O bit de sinal é o bit mais à esquerda possível. Assim, o maior inteiro positivo com 8 bits é 01111111 , ou seja,

$$\boxed{0} \times 2^7 + \boxed{1} \times 2^6 + \boxed{1} \times 2^5 + \boxed{0} \times 2^4 + \boxed{1} \times 2^3 + \boxed{1} \times 2^2 + \boxed{1} \times 2^1 + \boxed{1} \times 2^0 = 127$$

A representação de números negativos nos computadores digitais é uma questão à parte que não vamos detalhar nesta aula. Só tenha em mente que o bit mais à esquerda deve ser 1 para números negativos. Um exemplo: a representação do -1 em 8 bits é 11111111.

Considerando um byte com o bit de sinal é possível representar então $2^8 = 256$ valores (de -128 a $+127$). Com 16 bits ou 2 bytes é possível representar 2^{16} valores (de -32768 a $+32767$) e, com uma palavra (conjunto de bits) de 32 bits, 2^{32} (de -2147483648 a $+2147483647$). Atualmente, boa parte dos computadores pessoais trabalham com palavras de 32 bits (embora já seja comum encontrar máquinas de 64 bits).

Na linguagem C, ao declararmos uma variável, o compilador reserva na memória o espaço necessário para representá-la. Como esse espaço é fixo (por exemplo, 32 bits para variáveis inteiras), é possível que durante uma computação o número de bits utilizado não seja suficiente para representar os valores necessários, e nesse caso, os resultados são, obviamente, inválidos.

Dependendo do maior/menor número que seu programa precisa representar, além de **int** você pode declarar variáveis inteiras como **char** (para usar palavras de 8 bits) – veja a aula sobre caracteres.

13.2 Representação de Números Reais

Uma variável do tipo real é uma variável que pode conter números nos quais existe dígitos significativos à direita do ponto decimal. Por exemplo, 3.2, 21.43 0.12, etc.

Na memória do computador não podemos armazenar $1/2$ bit (apenas os zeros e uns). Como então representar um número fracionário, ou real? A representação é análoga à notação científica, feita em ponto flutuante da seguinte forma:

$$0.x_1x_2x_3...x_k \times B^e$$

onde $x_1x_2x_3...x_k$ é a mantissa (os k dígitos mais significativos do número), B é a base e e é o expoente (através do qual se determina a posição correta do dígito mais significativo do número em ponto flutuante). Essa notação permite a representação de uma faixa bastante grande de números, como por exemplo:

Número	Notação Científica	Mantissa	Base	Expoente
1000000000	0.1×10^{10} ou 1E9	1	10	10
123000	0.123×10^6 ou 1.23E5	123	10	6
456.78	0.45678×10^3 ou 4.5678E2	45678	10	3
0.00123	0.123×10^{-2} ou 1.23E$-$3	123	10	-2

Note que o "ponto flutuante" corresponde à posição do ponto decimal, que é "ajustado" pelo valor do expoente, e que nesse exemplo a mantissa, a base e o expoente são agora números inteiros. Uma notação semelhante pode ser utilizada para números binários, e reservamos espaço na memória (ou bits de uma palavra) para armazenar a mantissa e o expoente (a base pode ser pré-determinada, 2 no caso dos computadores digitais). Assim, a representação de um número real com 32 bits poderia usar 24 bits para a mantissa e 7 para o expoente. Como você mesmo pode observar, da mesma forma que os inteiros, os números em ponto flutuante são armazenados como um conjunto fixo de bytes, de modo que a sua precisão é limitada.

Dessa forma, o computador é uma máquina com capacidade de armazenamento finita. Assim, o conjunto de números que podem ser representados no tipo real não é o mesmo conjunto de números reais da matemática, e sim um subconjunto dos números racionais.

13.3 Variável Tipo Real

Os tipos de dados inteiros servem muito bem para a maioria dos programas, contudo alguns programas orientados para aplicações matemáticas frequentemente fazem uso de **números reais** (ou em **ponto flutuante**). Para este tipo de dados, em C, podemos utilizar os tipos **float** e **double**.

A diferença entre estes dois tipos é que no tipo de dado **double**, podemos representar uma quantidade maior de números reais que no tipo **float**. O tipo **double** usa 8 bytes para guardar um número em ponto flutuante (53

bits para a mantissa e 11 para o expoente); enquanto o **float** usa 4 bytes (24 bits para a mantissa e 8 para o expoente).

Os valores do tipo **float** são números que podem, em valor absoluto, serem tão grandes com 3.4×10^{38} ou tão pequenos quanto 3.4×10^{-38}. O tamanho da mantissa para este tipo de dado é 7 dígitos decimais e são necessários 4 bytes de memória para armazenar um valor deste tipo.

Os valores **double** são números que podem, em valor absoluto, serem tão grandes com 1.7×10^{308} ou tão pequenos quanto 1.7×10^{-308}. O tamanho da mantissa para este tipo de dado é 15 dígitos decimais e são necessários 8 bytes de memória para armazenar um valor deste tipo.

Assim, o tipo **float** tem uma precisão de 6 a 7 casas decimais com o expoente variando entre 10^{-37} a 10^{+38} e o tipo **double** uma precisão de 15 casas decimais com expoente variando entre 10^{-308} a 10^{+308} ocupando um espaço maior para armazenar um valor na memória. Isto significa que um número como 123456.78901234 será armazenado apenas como $1.234567E6$ em uma variável do tipo **float** ficando o restante além da precisão possível para a representação.

Neste curso, vamos usar o tipo **float**.

A forma para declarar uma variável do tipo **float** é a mesma para declarar variáveis do tipo **int**; só que em vez de usar a palavra chave **int**, deve-se usar a palavra **float**:

float <nome_da_variavel>;

Exemplo: declaração de uma variável do tipo **float** de nome "r"

float r;

Se você quiser declarar várias variáveis, é possível fazer da seguinte forma:

float <nome_da_variavel_1>, <nome_da_variavel_2>, <nome_da_variavel_3>,..., <nome_da_variavel_n>;

Exemplo: declaração de duas variáveis do tipo **float** "r1" e "r2".

float r1, r2;

13.4 Leitura de um Número Real pelo Teclado

Como vimos nas aulas passadas, para ler um número inteiro pelo teclado, nós usamos o "%d" dentro do comando scanf. Assim, para ler um inteiro x fazemos:

```
1       int x;
2
3       printf ("Entre com um numero inteiro x > 0: ");
4       scanf  ("%d", &x);
```

Para ler um número real pelo teclado, você deve utilizar "%f" dentro do comando scanf.

Para mostrar um exemplo, considere o seguinte trecho de programa que lê um número real:

```
1       float x;
2
3       printf ("Entre com um número real: ");
4       scanf  ("%f", &x);
```

13.5 Impressão de Números Reais

Como vimos nas aulas passadas, para imprimir um número inteiro na tela, nós usamos o "%d" dentro do comando `printf`. Assim, para imprimir um inteiro `x` fazemos:

```
1       int x;
2
3       printf ("Entre com um numero x > 0: ");
4       scanf  ("%d", &x);
5
6       printf ("Número lido foi = %d\n", x);
```

Para imprimir um número real na tela, nós podemos usar o "%f" dentro do comando `printf`:

```
1       float x;
2
3       printf ("Entre com um número real: ");
4       scanf  ("%f", &x);
5
6       printf ("Numero Digitado = %f\n", x);
```

É possível imprimir números reais ainda de outras formas:

%e	imprime um valor real em notação científica
%f	imprime um valor real em notação decimal
%g	imprime um valor real na notação científica ou decimal, como for mais apropriada

Veja o seguinte exemplo:

```
1       #include <stdio.h>
2
3       int main () {
4         float  f = 3.141592654;
5
6         printf("formato e: f=%e\n", f);
7         printf("formato f: f=%f\n", f);
8         printf("formato g: f=%g\n", f);
9
10        return 0;
11      }
```

A saída desse programa é:

```
formato e: f=3.141593e+000
formato f: f=3.141593
formato g: f=3.14159
```

13.5.1 Formatação de Impressão de Números Reais

Muitas vezes, para facilitar a visualização dos resultados, é necessário formatar os dados na saída do programa. Tanto o formato `%d` quanto o `%f` podem ser formatados no sentido de reservar um número de dígitos para impressão. Para usar formatação, você pode colocar entre o `%` e o caractere definindo o tipo (`d` ou `f`) o seguinte:

- **um sinal de menos**: especifica ajustamento à esquerda (o normal é à direita).
- **um número inteiro**: especifica o tamanho mínimo do campo. Se o número a ser impresso ocupar menos espaço, o espaço restante é preenchido com brancos para completar o tamanho desejado, mas se o número ocupar um espaço maior, o limite definido não é respeitado.
- **um ponto seguido de um número**: especifica o tamanho máximo de casas decimais a serem impressos após o ponto decimal. A precisão padrão para números reais é de 6 casas decimais.

Exemplos:

Considere a variável real `cempi = 314.159542` e veja como ela pode ser impressa usando diferentes formatos (as barras verticais facilitam a visualização):

```
1       float cempi = 314.159542;
2
3       printf("cempi = |%-8.2f|\n", cempi);
4       printf("cempi = |%8.2f|\n", cempi);
5       printf("cempi = |%8.4f|\n", cempi);
6       printf("cempi = |%8.4f|\n", cempi * 1000);
```

A impessão na tela fica:

```
cempi = |314.16  |
cempi = |  314.16|
cempi = |314.1595|
cempi = |314159.5313|
```

Observe que 8 casas incluem o ponto decimal, e são suficientes para os primeiros 3 `printf`'s. No último `printf` esse limite não é obedecido, pois o número a ser impresso ocupa um lugar maior que 8. Observe também que o tipo **float** perde precisão em torno da sexta casa decimal, daí os últimos digitos de `cempi * 1000` não estarem corretos.

13.6 Escrita de Números Reais

Números em ponto flutuante podem ser definidos de diversas formas. A mais geral é uma série de dígitos com sinal, incluindo um ponto decimal, depois um '`e`' ou '`E`' seguido do valor do expoente (a potência de dez) com sinal. Por exemplo: `-1.609E-19` e `+6.03e+23`. Essas cIonstantes podem ser utilizadas em expressões como por exemplo:

```
1       float x = 3.141595426;
2       float y = 1.23e-23;
3 I
```

Na definição de números reais pode-se omitir sinais positivos, a parte de expoente e a parte inteira ou fracionária. Exemplos:

- 3.14159
- .2
- 4e16
- .8e−5
- 100

Não se deve usar espaços dentro de um número em ponto flutuante: O número `3.34␣E+12` está errado.

13.7 Expressões Aritméticas Envolvendo Reais

Ao utilizarmos números reais em nossos programas, é comum misturar números e variáveis inteiras com reais em nossas expressões aritméticas. Para cada operador (+, -, *, /, etc) da expressão, o compilador precisa decidir se a operação deve ser realizada como inteira ou como real, pois como a forma de representação de inteiros e reais é diferente, as operações precisam ser feitas usando a mesma representação. A regra básica é, se os operandos tiverem tipos diferentes, a operação é realizada usando o "maior" tipo, ou seja, se um dos operandos for real, o resultado da operação é real, caso contrário, a operação é inteira.

13.7.1 Observação quanto à Divisão

```
1       int i,j;
2       float y;
3
4       i = 5 / 3; /* divisão inteira e o resultado é 1 (5 e 3 são inteiros) */
5       y = 5 / 3; /* divisão inteira e o resultado é 2.0 (y é real) */
6       y = 5.0 / 2; /* divisão tem como resultado 2.5 (o numerador é real) */
7       y = 5 / 2.0; /* divisão tem como resultado 2.5 (o denominador é real) */
8
9       y = i / 2; /* divisão inteira (i e 2 são inteiros) */
10      y = i / 2.0; /* divisão em ponto flutuante (denominador real) */
11      y = i / j; /* divisão inteira (i e j são inteiros) */
12      y = (1.0 * i) / j; /* divisão em ponto flutuante (numerador real) */
13      y = 1.0 * (i / j); /* divisão inteira (i e j são inteiros) */
14      i = y / 2; /* parte inteira da divisão em i (divisão real, mas i é inteiro) */
15 I
```

Veja a saída do programa abaixo e tente entender o que acontence no primeiro e no segundo printf:

```
1       #include <stdio.h>
2
3       int main () {
4         int i=4;
5         int j=5;
6         int k;
7         float f = 5.0;
8         float g;
9
10        k = 6*(j/i); /* variável inteira k recebe resultado de expressão inteira */
11        g = 6*(f/i); /* variável real g recebe resultado de expressão real */
12        printf("1: k=%d g=%f\n", k, g);
13
14        g = 6*(j/i); /* variável real g recebe resultado de expressão inteira */
15        k = 6*(f/i); /* variável inteira k recebe resultado de expressão real */
16        printf("2: k=%d g=%f\n", k, g);
17
18        return 0;
19      }
20 I
```

A saída dos `printf`'s é:

```
1: k=6 g=7.500000
2: k=7 g=6.000000
```

Lembre-se que em uma atribuição, cada expressão é calculada (lado direito) e o resultado é depois armazenado na variável correspondente, definida no lado esquerdo da atribuição. Nas atribuições antes do primeiro `printf`,

o tipo da expressão é o mesmo da variável, mas nas atribuições seguintes, os tipos são diferentes. Observe portanto que o tipo da variável que recebe a atribuição NÃO influencia a forma de calcular as expressões. Após o cálculo, o resultado é convertido ao tipo da variável (ou seja, inteiro 6 passa a real 6.0 e real 7.5 passa a inteiro 7). É possível forçar a mudança de tipos de um termo dentro de expressão através de definições explícitas conhecidas como **type casting**. Observe o exemplo abaixo:

```
1       #include <stdio.h>
2
3       int main () {
4         int i=4;
5         int j=5;
6         int k;
7         float f = 5.0;
8         float g;
9
10        /* variável inteira k recebe resultado de expressão inteira */
11        k = 6*(j/i);
12
13        /* variável real g recebe resultado de expressão inteira, */
14        /* pois a variável f foi explicitamente convertida para o tipo int */
15        g = 6*((int)f/i);
16
17        printf("1: k=%d g=%f\n", k, g);
18
19
20        /* o número 6 é promovido a float, e portanto o resultado é real */
21        /* uma forma mais simples seria definir o número 6 como 6.0 */
22        g = (float)6*j/i;
23
24
25        /* variável inteira k recebe a parte inteira do resultado da expressão real */
26        k = 6*(f/i);
27
28        printf("2: k=%d g=%f\n", k, g);
29
30        return 0;
31      }
32 I
```

13.8 Exercício

Dado um natural n, determine o número harmônico H_n definido por

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Solução Comentada:

A somatória indica que precisamos realizar as seguites operações:

$$H_n = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

Imediatamente, já podemos observar que precisamos fazer n somas, gerando uma sequência de inteiros de 1 a n. Para isso, precisamos de uma repetição que tem a seguinte estrutura:

```
1       i = 1;
2       while (i<=n) {
3         soma = soma + 1 / i;
4         i = i + 1;
5       }
6 I
```

ou usando a forma mais sucinta com o comando **for**:

```
1       for (i=1; i<=n; i++) {
2         soma = soma + 1 / i;
3       }
4 I
```

Observe que i pode ser uma variável inteira e soma PRECISA ser uma variável real. Por que não utilizamos então todas as variáveis reais? Por várias razões. Uma delas é a consistência, já que o número de termos da soma é inteiro, faz sentido (fica mais fácil de entender) se a variável for inteira, e devido ao desempenho do seu programa, pois as operações com inteiros são realizadas mais rapidamente pelo computador que as operações em ponto flutuante (real).

A solução final para esse programa seria:

```
1       #include <stdio.h>
2       int main () {
3
4         float soma = 0;
5         int i;
6
7         printf("Entre com o valor de n>0: ");
8         scanf("%d", &n);
9
10        for (i = 1; i<=n; i++) {
11          soma = soma + 1 / i;
12        }
13
14        printf("o número harmonico H%d = %f\n", n, soma);
15
16        return 0;
17      }
18 I
```

Aparentemente, essa solução está correta, porém, ela possui um erro difícil de notar. Teste esse programa, exatamente como está escrito acima, e verifique que a saída do programa é sempre 1, para qualquer $n > 0$. Por que?

Vimos que o compilador decide, para cada operação, se ela deve ser realizada como inteiro e como real, dependendo do tipo dos operandos envolvidos. Veja com atenção a linha:

```
soma = soma + 1 / i;
```

Devido a precedência dos operadores, a divisão é feita antes da soma. Como os operandos 1 e i são ambos inteiros, o resultado da divisão também é inteiro, ou seja, quando $i > 1$, o resultado é sempre 0, daí o resultado ao final sempre ser 1. Para resolver esse erro, basta explicitamente colocar a constante real 1.0 ou colocar um casting antes do número inteiro 1 ou na variável inteira i, como abaixo:

```
soma = soma + 1.0 / i;  |  soma = soma + (float) 1 / i;  |  soma = soma + 1 / (float) i;
```

Uma solução final para esse programa seria:

```
1       #include <stdio.h>
2       int main () {
3
4         float soma = 0;
5         int i;
6
7         printf("Entre com o valor de n>0: ");
8         scanf("%d", &n);
9
10        for (i = 1; i<=n; i++) {
11          soma = soma + 1.0 / i;
12        }
13
14        printf("o número harmonico H%d = %f\n", n, soma);
15
16        return 0;
17      }
18 I
```

13.9 Exercícios recomendados

1. Dado um número inteiro $n > 0$, calcular o valor da soma

$$s_n = 1/n + 2/(n-1) + 3/(n-2) + 3/(n-2) + \ldots + n/1.$$

2. Dado um número real x e um número real $epsilon > 0$, calcular uma aproximação de e^x através da seguinte série infinita:

$$e^x = 1 + x + x^2/2! + x^3/3! + \ldots + x^k/k! + \ldots$$

   Inclua na aproximação todos os termos até o primeiro de valor absoluto (módulo) menor do que $epsilon$.