



# Estruturas de Dados

Prof. Guilherme N. Ramos

## 1 Estruturas de Dados

Todo programa é a implementação de um algoritmo computacional, e manipula dados por definição (ao rodar em um computador de *programa armazenado*), e variáveis e constantes são os objetos de dados básicos manipulados em um programa [1]. As operações que podem ser realizadas nos dados são definidas por seus tipos, e os mais comuns são numéricos, simbólicos e lógicos.





Os dados são armazenados em bits (*binary digits*) na memória, que pode ser vista como um conjunto ordenado de bits. Cada bit representa um estado binário cujo valor é:

“ligado” é representado pelo símbolo 1, e

“desligado” é representado pelo símbolo 0.

Um bit define apenas 2 estados distintos (0/1) e mutuamente exclusivos [3], mas se considerarmos 2 bits são 4 estados (00/01/11/10). 3 bits definem 8 estados, e assim sucessivamente; então sabe-se que  $n$  bits possibilitam  $2^n$  estados distintos. Para facilitar, há uma nomenclatura específica para lidar com a quantidade de bits: 8 bits compõem 1 byte, e  $10^6$  bytes são 1 MB.

Considerando 1 byte, pode-se definir 256 estados diferentes, mas *o que representa cada estado?* Essa é uma decisão arbitrária extremamente importante, pois dados *diferentes* podem ser representados por um mesmo [conjunto de] byte[s]. Por exemplo:

	$\mathbb{N}$	$\mathbb{Z}$	Letra	Imagem	...
00000000	0	0	A		...
00000001	1	1	B		...
00000010	2	2	C		...
⋮	⋮	⋮	⋮	⋮	⋮
11111111	255	-127	?		...

Na memória do computador, a representação física dos dados é uma só: *binária*<sup>1</sup>. Mas a *interpretação* dos bits define a informação. Assim como dados diferentes podem ser armazenados como um mesmo conjunto de bits, conjuntos diferentes de bits podem ser interpretados como o mesmo dado.

## 2 Sistemas Numéricos

Bits, assim como algarismos, podem representar números pelo sistema numérico posicional que é baseado na soma ponderada dos valores dos símbolos da base, de acordo com sua posição. Por exemplo, na base decimal o valor do número representados pelos símbolos 123 é dado por:

$$100 + 20 + 3 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

O valor de qualquer número em uma representação posicional depende de cada algarismo que o compõe e de sua posição. Os algarismos dependem da base numérica, mas o valor em qualquer base pode ser facilmente obtido com a seguinte fórmula:

$$a_n a_{n-1} \cdots a_2 a_1 a_0 = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \cdots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0$$

<sup>1</sup>Há 10 tipos de pessoas: as que percebem código binário e as que não.

Onde  $b$  é a base numérica e  $a_i$  é o algarismo na  $i$ -ésima posição do número, sendo  $0 \leq a_i < b$ . Em computação, as bases mais utilizadas são *hexadecimal*, com os algarismos  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ , *decimal*, com  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , *octal*, com  $\{0, 1, 2, 3, 4, 5, 6, 7\}$ , e *binária*, com  $\{0, 1\}$ . Assim, tem-se:

$$7B_{16} = 123_{10} = 173_8 = 1111011_2$$

## 2.1 Números Naturais

Os números naturais são facilmente representados pelo sistema de numeração binário. Por exemplo, 3 bits podem representar:

$$000 = 0 \cdot 2^2 + 0 \cdot 2 + 0 = 0$$

$$001 = 0 \cdot 2^2 + 0 \cdot 2 + 1 = 1$$

$$010 = 0 \cdot 2^2 + 1 \cdot 2 + 0 = 2$$

$$011 = 0 \cdot 2^2 + 1 \cdot 2 + 1 = 3$$

$$100 = 1 \cdot 2^2 + 0 \cdot 2 + 0 = 4$$

$$101 = 1 \cdot 2^2 + 0 \cdot 2 + 1 = 5$$

$$110 = 1 \cdot 2^2 + 1 \cdot 2 + 0 = 6$$

$$111 = 1 \cdot 2^2 + 1 \cdot 2 + 1 = 7$$

É importante notar que, como são armazenados na memória, os valores dos números são limitados pela quantidade de bits utilizada.

## 2.2 Números Inteiros

Como os naturais, é fácil representar os números inteiros em binário com a mesma lógica posicional. Entretanto, estes números podem ter valores negativos, e é preciso considerar esta informação.

A solução encontrada foi utilizar um bit (arbitrariamente, sempre o mais a esquerda) para indicar o sinal do número: 0 implica que o número é *positivo*, 1 que é negativo. Há três formas distintas de interpretar estes números.

Sinal e Magnitude considera o bit mais a esquerda como indicador de sinal e os bits restantes diretamente pelo sistema posicional (como um número natural).

$$000 \rightarrow +(0 \cdot 2 + 0) = +0$$

$$001 \rightarrow +(0 \cdot 2 + 1) = +1$$

$$010 \rightarrow +(1 \cdot 2 + 0) = +2$$

$$011 \rightarrow +(1 \cdot 2 + 1) = +3$$

$$100 \rightarrow -(0 \cdot 2 + 0) = -0$$

$$101 \rightarrow -(0 \cdot 2 + 1) = -1$$

$$110 \rightarrow -(1 \cdot 2 + 0) = -2$$

$$111 \rightarrow -(1 \cdot 2 + 1) = -3$$

Complemento de um considera o bit mais a esquerda como indicador de sinal. Se *positivo*, os bits restantes indicam o valor pelo sistema posicional; se *negativo*, é preciso inverter todos os bits antes de considerar o sistema posicional.

$$\begin{aligned}
 000 &\rightarrow +00 = +(0 \cdot 2 + 0) = +0 \\
 001 &\rightarrow +01 = +(0 \cdot 2 + 1) = +1 \\
 010 &\rightarrow +10 = +(1 \cdot 2 + 0) = +2 \\
 011 &\rightarrow +11 = +(1 \cdot 2 + 1) = +3 \\
 100 &\rightarrow -11 = -(1 \cdot 2 + 1) = -3 \\
 101 &\rightarrow -10 = -(1 \cdot 2 + 0) = -2 \\
 110 &\rightarrow -01 = -(0 \cdot 2 + 1) = -1 \\
 111 &\rightarrow -00 = -(0 \cdot 2 + 0) = -0
 \end{aligned}$$

Por fim, a forma mais utilizada é complemento de dois, que considera o bit mais a esquerda como indicador de sinal. Se *positivo*, os bits restantes indicam o valor pelo sistema posicional; se *negativo*, é preciso inverter todos os bits e incrementar em 1 o resultado antes de considerar o sistema posicional. Embora pareça mais complicada, esta forma tem uma série de vantagens (como quantidade de valores distintos e facilidade na computação de operações matemáticas).

$$\begin{aligned}
 000 &\rightarrow 000 \rightarrow +000 = +((0 \cdot 2^2 + 0 \cdot 2 + 0) = +0 \\
 001 &\rightarrow 001 \rightarrow +001 = +((0 \cdot 2^2 + 0 \cdot 2 + 1) = +1 \\
 010 &\rightarrow 010 \rightarrow +010 = +((0 \cdot 2^2 + 1 \cdot 2 + 0) = +2 \\
 011 &\rightarrow 011 \rightarrow +011 = +((0 \cdot 2^2 + 1 \cdot 2 + 1) = +3 \\
 100 &\xrightarrow{inv} 011 \xrightarrow{+1} -100 = -(1 \cdot 2^2 + 0 \cdot 2 + 0) = -4 \\
 101 &\xrightarrow{inv} 010 \xrightarrow{+1} -011 = -(0 \cdot 2^2 + 1 \cdot 2 + 1) = -3 \\
 110 &\xrightarrow{inv} 001 \xrightarrow{+1} -010 = -(0 \cdot 2^2 + 1 \cdot 2 + 0) = -2 \\
 111 &\xrightarrow{inv} 000 \xrightarrow{+1} -001 = -(0 \cdot 2^2 + 0 \cdot 2 + 1) = -1
 \end{aligned}$$

É interessante notar que, dependendo da interpretação, um mesmo conjunto de bits pode ter valores numéricos diferentes (ex: 111). Além disso, o conjunto dos números inteiros é infinito, mas a memória tem apenas uma quantidade finita de bits. Considere o código para calcular a raiz de um número inteiro:

0-raiz2-4.c

```

1 int erro(int r, int n) {
2     return abs(r*r - n);
3 }
4
5 int raiz2(int n) {
6     int r = n/2;
7
8     if(n < 2)
9         return (n < 0 ? -1 : n);
10
11     while(erro(r, n) > r)
12         r = (r+(n/r))/2;
13
14     return r;
15 }

```

Sabendo que o tipo `int` usa 32 bits (complemento de 2), o que acontece quando se tenta calcular a raiz de  $10^{20}$ ? (veja `2-limites.c`, e pense no que ocorre na linha 2)

## 2.3 Números Reais

Os números reais também podem ser representados como binários pelo sistema posicional, basta estender a lógica para os valores não inteiros. Por exemplo:

$$13,125 = 1 \cdot 10^1 + 3 \cdot 10^0 + 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

Este valor pode ser representado com a mesma lógica, agora em binário:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1101,001$$

A representação com ponto fixo define que, dada uma quantidade  $Q$  de bits para representar o número, há uma quantidade fixa  $m$  de bits que armazenam a parte inteira e outra quantidade  $f$  que armazena a parte fracionária do número (tais que  $Q = m + f$ ). Por exemplo, supondo  $Qm.f = Q5.3$ , o número 13,125 seria representado pelos bits 01101001 (“significando” 01101,001). Supondo  $Q4.4$ , seriam os bits 11010010.

Já a representação em ponto flutuante aproveita as vantagens da notação científica (por exemplo,  $13,125 = 0,13125 \cdot 10^2$ ). Da mesma forma, qualquer número binário pode ser representado como  $m \cdot 2^e$ , sendo  $m$  o valor da *mantissa* e  $e$  o *expoente*.

A ideia é simples: define-se uma quantidade fixa de bits para armazenar a mantissa, e o restante para armazenar o expoente. Esta representação oferece maior flexibilidade e alcance para números reais (comparada ao ponto fixo). O padrão dos computadores modernos é o IEEE 754, que considera precisão simples (32 bits) e dupla (64 bits), e o define o valor armazenado pela equação:

$$(-1)^{sinal} \cdot (1 + mantissa) \cdot 2^{expoente - offset}$$

No caso da precisão simples, o primeiro bit (mais a esquerda) indica o sinal, os 8 bits seguintes o expoente, e os 23 bits restantes a mantissa (parte fracionária); o *offset* tem valor 127 (por que?). Considerando os 32 bits 1 01111110 100000000000000000000000, tem-se:

$$\begin{aligned} & (-1)^1 \cdot 1,1 \cdot 2^{126-127} \\ &= -1,1 \cdot 2^{-1} \\ &= -0,11 \\ &= -(1 \cdot 2^{-1} + 1 \cdot 2^{-2}) \\ &= -(0,5 + 0,25) \\ &= -0,75 \end{aligned}$$

sinal		expoente								mantissa																						
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

O mesmo processo é realizado para precisão dupla, mas neste caso o expoente tem 11 bits, a mantissa 52 bits, e o *offset* é 1023. Este tipo de representação possibilita o uso de números muito pequenos/grande (veja `2-limites.c`). Como exercício, tente definir a representação do valor 576,73 em ponto flutuante de precisão simples.

O uso de ponto flutuante oferece diversas vantagens, principalmente a representação de valores absolutos muito grandes ou pequenos [3]; mas há um “pequeno problema” (veja `3-precisao_float.c`). Suponha o código a seguir, com cuja representação do tipo `float` é no padrão IEEE 754 (precisão simples). Qual mensagem seria mostrada por sua execução?

```

1  float soma = 0;
2  for(i = 0; i < 10; ++i)
3      soma += 0.1;
4
5  if(soma == 1)
6      printf("soma == 1\n\n");
7  else
8      printf("soma != 1\n\n");

```

Se você entendeu direito a representação na memória, deve ter percebido que o comportamento esperado (e correto!) é que a mensagem seja: “soma != 1”. Se não concorda, tente representar o valor 0,1 em ponto flutuante. Se ainda assim não estiver muito claro, analise este código: 3-precisao\_float.c.

Esta imprecisão tem uma série de implicações. A mais clara é que você não deve comparar diretamente variáveis com este tipo, pois valores que “deveriam” ser iguais não são [necessariamente]. A solução é considerar uma tolerância aceitável entre os valores. Outra implicação é que é preciso muita atenção a possibilidade de acúmulo de erro, pois podem ser realmente significativos.

A Linguagem C oferece diversos tipos numéricos, cujos tamanhos (quantidade de bits utilizados para armazenagem) *depende da implementação*; isso permite que o programador explore as vantagens do hardware [2]. Isso pode ser um problema, pois afeta a portabilidade do código.

## 2.4 Símbolos

Símbolos são uma forma extremamente versátil de comunicar informações e - claro - podem ser representados por bits. O alfabeto define um pequeno conjunto de símbolos que, juntos, podem expressar quase tudo que se deseja.

Uma vez estabelecido um padrão de *codificação de caracteres* (associação [arbitrária] de bits a certos caracteres), pode-se armazenar estes símbolos na memória (e recuperá-los). Há diversas formas de se codificar caracteres: EBCDIC, Unicode, ente outros.

Nesta disciplina, o foco é a representação em ASCII, uma forma extremamente compacta e a mais utilizada no padrão ANSI. Neste contexto, um `char` é um pequeno inteiro, de modo que pode ser livremente usado em expressões aritméticas [1]

## 3 Ponteiros

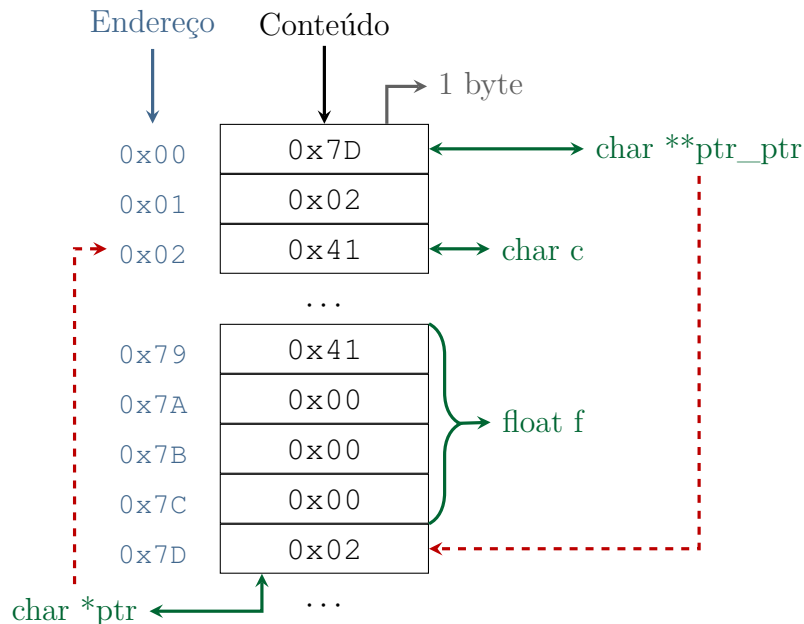
Cada variável declarada ocupa um espaço na memória, conforme seu tipo, e nome da variável é apenas uma forma “amigável” de lidar com o endereço deste espaço. Ponteiro (ou *apontador*) é um tipo de dado que armazena um *endereço de memória*, possibilitando leitura e escrita deste endereço. São muito usado na linguagem C por serem, as vezes, a única forma de expressar uma computação [1] (e geralmente tornarem o código mais compacto e eficiente). Simplificando, para um tipo T, a variável `T* p` é do tipo *ponteiro para T* e pode armazenar o endereço de um objeto do tipo T [2].

Em linguagem C, um ponteiro é declarado da seguinte forma: `tipo *identificador`. Por exemplo:

```

1  int*   ptr_int;      /* ponteiro para inteiro */
2  float* ptr_float;    /* ponteiro para real */
3  char*  ptr_char;     /* ponteiro para caractere */
4
5  int**  ptr_ptr_int;  /* ponteiro para (ponteiro para inteiro) */

```



O endereço de memória identifica o espaço físico na memória dos bytes que armazenam a informação. Por exemplo, na figura acima, o *endereço* 0x02 indica o byte identificado como *c*. *c* é uma variável do tipo `char`, que neste exemplo ocupa 1 byte de memória. Ao analisar o *conteúdo* de 0x02, vê-se que o valor armazenado é 0x41 (símbolo ‘A’ na tabela ASCII).

*Atenção a diferença conceitual entre **endereço** e **conteúdo**. O endereço indica a localização na memória (onde está armazenado), o conteúdo indica o valor dos bits (o que está armazenado).*

Da mesma forma, o byte no endereço 0x79 é identificado como *f*, e como a variável é do tipo `float`, o computador sabe que ela ocupa (neste exemplo) 4 bytes de memória (0x41000000 = 8).

O byte no endereço 0x7D é identificado como *ptr*, do tipo `char *` (ponteiro para `char`) que, neste exemplo, ocupa 1 byte. Por ser um ponteiro, o conteúdo deste identificador é interpretado como um número natural que *aponta* para um endereço de memória (no exemplo, para o endereço 0x02). Por ser um ponteiro para `char`, o conteúdo deste endereço é tratado como `char`.

Sabendo o endereço de memória de acesso aleatório, pode-se acessar diretamente a posição indicada pelo ponteiro e verificar seu conteúdo e dizer, por exemplo, *qual o caractere armazenado no endereço apontado por ptr?*

Uma das vantagens de se utilizar ponteiros é lidar com endereços de memória, de forma “independente” do tipo (os bytes têm endereços representados da mesma forma, independentemente do que armazenam). Por exemplo, assim como *ptr*, *ptr\_ptr* identifica 1 byte e armazena um endereço de memória; mas por ser um ponteiro para ponteiro, *ptr\_ptr*, aponta para um endereço de memória que também armazena um endereço de memória, no caso um *ponteiro para char*.

#### 0-ponteiro.c

```
1 char c = 'A';
2 char* ptr = &c; /* Armazena o endereço de c */
3
4 /* O conteúdo de c é: */
5 printf("  c = %c\n", c);
6 /* O conteúdo de ptr é: */
7 printf(" ptr = %p\n", ptr);
8 /* O conteúdo do endereço apontado por ptr é: */
9 printf(" *ptr = %c\n", *ptr);
10 /* O endereço de ptr é: */
11 printf("&ptr = %p\n", &ptr);
```

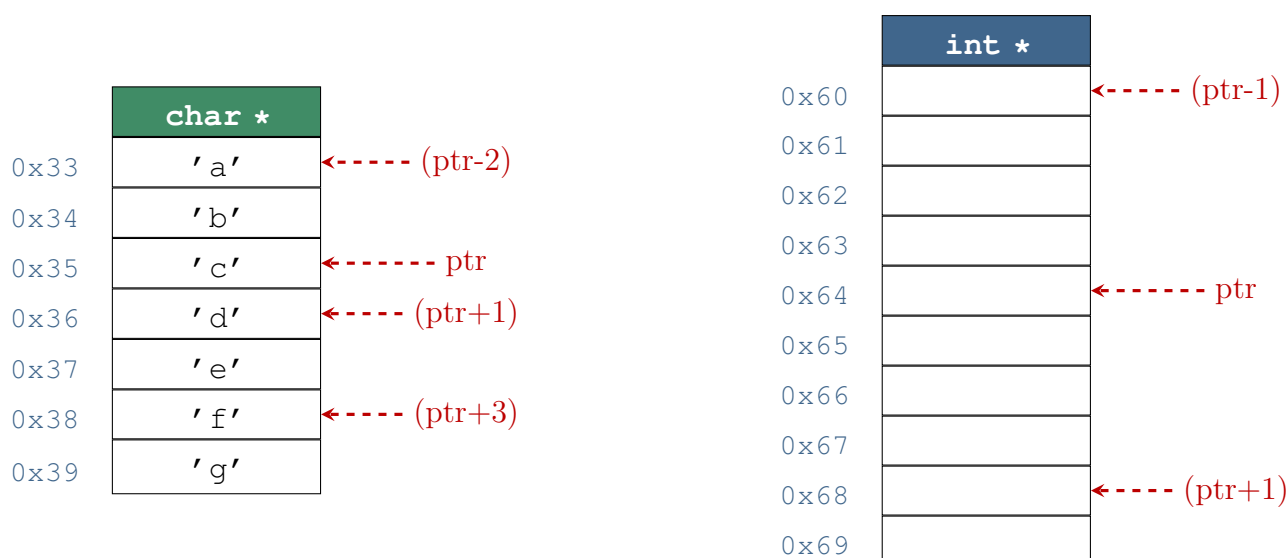
#### 1-tipos.c

```
1 int i = 10;
2 char c = 'A';
3 float f = 1.5;
4 double d = 3.14;
5 int* pi = &i;
6 char* pc = &c;
7 float* pf = &f;
8 double* pd = &d;
```

Na linguagem C, o conteúdo de uma variável é acessado por seu identificador, e o mesmo é válido para ponteiros. Entretanto, muitas vezes deseja-se acessar o conteúdo do endereço apontado pelo ponteiro, para tanto utiliza-se o operador unário `*`. Para obter o endereço do conteúdo de um identificador, utiliza-se o operador unário `&`.

A linguagem Python, por projeto, manipula objetos de forma diferente de C. Em ambas, os argumentos são passados por valor, mas dependendo do tipo de objeto (mutável/imutável), a manipulação dentro do escopo de uma função tem ou não efeito em um escopo externo. O comportamento de “passagem por referência” pode ser obtido de outras formas.

A linguagem C é fortemente tipada, portanto ao declarar um ponteiro já se sabe o tipo de dado a ser apontado e, conseqüentemente, a quantidade de bytes que cada conteúdo ocupa. Isso têm efeitos interessantes na aritmética de ponteiros. Por armazenarem números naturais, pode-se adicionar ou subtrair de ponteiros para acessar outros elementos na memória, e conforme a tipagem, sabe-se exatamente quantos bytes equivalem a “uma unidade” do tipo. Desta forma, alterar o conteúdo de um ponteiro para char em uma unidade seria o equivalente a deslocá-lo 1 byte, mas seriam 4 no caso de um ponteiro para int (veja 2-ponteiro.c).



*É preciso muito cuidado ao utilizar aritmética de ponteiros, pois você pode lidar com uma referência inválida de memória.*

Como qualquer outro tipo, ponteiros podem ser utilizados como argumentos de funções. Um ponteiro dado como argumento é armazenado no escopo da função, mas seu conteúdo pode indicar um endereço de memória de outro escopo, possibilitando aplicações muito mais interessantes. Compare o código de troca a seguir com o visto em 3-escopo.c.

apc\_ponteiro.h

```
1 /* Troca os conteúdos dos inteiros. */
2 void troca_i(int* a, int* b) {
3     int aux = (*a);
4     (*a) = (*b);
5     (*b) = aux;
6 }
```

Esta versão de `troca_i` funciona como esperado, pois embora tenha suas variáveis locais e seu escopo, pode acessar diretamente os endereços de outros escopos para manipular a memória. É assim que as funções `scanf` e `printf` conseguem realizar suas tarefas.

Além disso, sabe-se que a comunicação de dados entre [sub]algoritmos é feita pela passagem de argumentos (se houver) e pelo valor de retorno (se houver). E que na linguagem C, o valor de retorno é sempre uma *saída* da função. Mas pode-se explorar o uso de ponteiro de forma que os argumentos fornecidos passados podem ser considerados de:

**entrada:** são recebidos e processados, mas não alterados;

**saída:** têm seus valores alterados para processamento após a execução da função; e

**entrada/saída:** fornece um valor à função e é alterado por sua execução.

Por exemplo,  $\{a, b, c\}$  de entrada e  $\{r1, r2\}$  de saída:

4-bhaskara.c

```
1 int bhaskara(double a, double b, double c,
2             double *r1, double *r2) {
3     double delta = b*b - 4*a*c;
4     int raizes_reais = (delta >= 0 ? 1 : 0);
5
6     if(raizes_reais) {
7         (*r1) = (-b + sqrt(delta))/2;
8         (*r2) = (-b - sqrt(delta))/2;
9     }
10
11     return raizes_reais;
12 }
```

Por fim, em um computador de programa armazenado, as instruções também são dados guardados na memória que têm endereços para acesso, então por que não usar ponteiros de funções?

5-funcao.c

```
1 #include "apc_numeros.h"
2
3 /* Chama a função dada usando os parâmetros dados (a,b) como
4  * argumentos. */
5 int chama(int (*func)(int, int), int a, int b) {
6     return func(a,b);
7 }
8
9 int main() {
10     int a = 1, b = 2;
11
12     printf("chama(max,%d,%d) = %d\n", a,b, chama(max_i,a,b));
13     printf("chama(min,%d,%d) = %d\n", a,b, chama(min_i,a,b));
14
15     a = 7;
16     printf("chama(max,%d,%d) = %d\n", a,b, chama(max_i,b,a));
17     printf("chama(min,%d,%d) = %d\n", a,b, chama(min_i,b,a));
18
19     return 0;
20 }
```

Esta ideia pode parecer um pouco confusa de início, mas possibilita programas extremamente interessantes. Por exemplo, o Método de Newton-Raphson serve para aproximar a raiz de um polinômio qualquer. A primeira implementação vista define as funções do polinômio ( $f$  e  $fp$ ) para encontrar a raiz quadrada, e a função `Newton_Raphson` as utiliza para realizar a tarefa.

O método numérico abstraiu a implementação das funções, de modo que funciona para quaisquer implementação de  $f$  e  $fp$ . Entretanto, a implementação exige que estas estejam bem definidas antes da compilação, prendendo a função de aproximação à definição do polinômio. Uma forma de superar esta limitação é utilizando ponteiros de função (`2-raizes.c`).

## Referências

- [1] Brian W. Kernighan and Dennis M. Ritchie. *C: a linguagem de programação padrão ANSI*. Campus, Rio de Janeiro, 1989.



- [2] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, Mass., 3. ed., 20. print edition, 2004.
- [3] Aaron M Tenenbaum, Yedidyah Langsam, and Moshe Augenstein. *Estruturas de dados usando C*. Pearson Makron Books, São Paulo (SP), 1995.