Capítulo 1: Introdução à Programação

Waldemar Celes e Roberto Ierusalimschy

29 de Fevereiro de 2012

1 Modelo de computador

O computador é uma máquina capaz de manipular informações processando seqüências de instruções. As informações manipuladas por um computador são dados que podem ser valores numéricos e/ou textos. As seqüências de instruções definem um *programa*. Programar um computador significa desenvolver uma seqüência de instruções que executa uma tarefa específica, produzindo o resultado esperado. Por se tratar de uma máquina que processa dados de uma maneira muito eficiente, o computador é uma valiosa ferramenta nos dias atuais. Identificamos o uso de computadores em quase todas as atividades do mundo moderno. Em especial, o computador é indispensável atualmente para exercermos nossas atividades na área técnico-científica.

Os componentes físicos que compõem um computador (a máquina) são chamados de *hardware*. Os programas que executam nas máquinas são chamados de *software*. Neste texto, vamos aprender como desenvolvemos software, discutindo técnicas de programação adequadas para obtenção de programas corretos e de boa qualidade. Embora não seja nosso objetivo estudar hardware, é importante identificarmos os principais elementos de um computador. Isso nos ajudará a compreender como o computador funciona e, por conseguinte, nos ajudará a elaborar programas melhores e corretos.

A Figura 1 ilustra os principais elementos de um computador típico. O canal de comunicação (conhecido como *bus*) representa o meio para a transferência de dados entre os diversos componentes. A unidade central de processamento (CPU, *central processing unit*) representa o "cérebro" do computador, sendo responsável por controlar todas as operações feitas por ele. Para que a CPU possa executar uma seqüência de comandos ou acessar uma determinada informação, é necessário que os comandos e os dados correspondentes estejam armazenados na memória principal. Na memória principal são armazenados, portanto, os programas e os dados manipulados pela CPU. A memória principal tem acesso randômico (RAM, *random access memory*), o que significa que a CPU pode acessar diretamente qualquer posição da memória. Esta memória não é permanente e, para um programa, os dados são armazenados enquanto o programa está sendo executado. Em geral, após o término do programa, os dados armazenados na memória principal são perdidos e a área correspondente ocupada na memória fica disponível para ser usada por outros programas. A memória principal é conhecida como área de armazenamento primário.

A área de armazenamento secundário é, em geral, representada por meios magnéticos (disco rígido, disquete, etc.). Esta memória secundária tem a vantagem de ser permanente. Os dados armazenados em disco permanecem disponíveis após o término dos programas. Esta memória tem um custo mais baixo do que a memória principal, porém o acesso aos dados é bem mais lento. Para que a CPU processe um dado armazenado na memória secundária, é necessário que o dado seja antes transferido para a memória principal.

Por fim, encontram-se os dispositivos de entrada e saída. Os dispositivos de entrada (por exemplo, teclado e mouse) permitem que os usuários forneçam dados para um programa, enquanto os dispositivos de saída permitem que um programa exiba os resultados computados, por exemplo em forma textual ou gráfica, usando monitores ou impressoras.

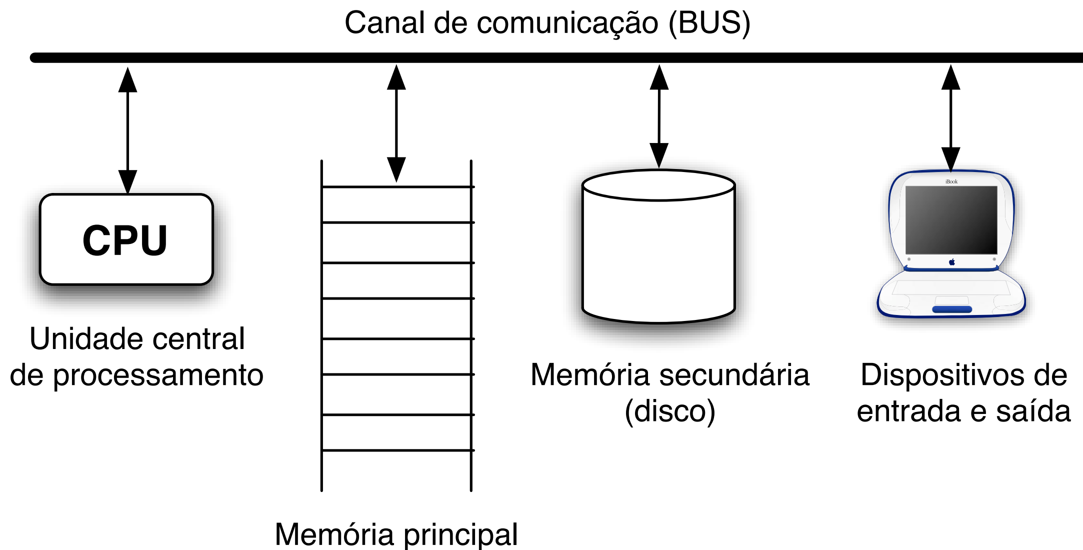

Figura 1: Principais elementos de um computador típico.

1.1 Computador hipotético simples

Para exemplificar alguns conceitos básicos de desenvolvimento de programas, vamos considerar a existência de um computador hipotético simplificado. Nosso computador é capaz de realizar apenas algumas poucas instruções básicas. Um programa para ser executado neste nosso computador deve ser composto por uma seqüência destas instruções. Neste computador, vamos imaginar que cada posição da memória serve para armazenar valores numéricos e é rotulada por um número. Assim, temos as posições rotuladas por `0`, `1`, `2`, e assim por diante. Nossa CPU também tem uma área de memória usada para armazenar, temporariamente, o valor resultante de uma operação. Chamaremos esta área de memória de *registrador*. As instruções disponíveis neste nosso computador hipotético são:

- **read** ***pos***: captura o valor fornecido pelo usuário via teclado e o armazena na posição da memória rotulada por *pos*.
- **write** ***pos***: escreve o valor armazenado na posição da memória rotulada por *pos* na tela do computador.
- **storeconst** ***num pos***: armazena o valor da constante numérica especificada (*num*) na posição de memória rotulada por *pos*.
- **add** ***pos1 pos2***: calcula a soma dos operandos, valores numéricos armazenados nas posições *op1* e *op2*. O resultado é armazenado no registrador.
- **sub** ***pos1 pos2***: calcula a diferença entre o primeiro e o segundo operando, armazenando o resultado no registrador.
- **mul** ***pos1 pos2***: calcula a multiplicação dos operandos, armazenando o resultado no registrador.
- **div** ***pos1 pos2***: calcula a divisão do primeiro pelo segundo operando, armazenando o resultado no registrador.
- **store** ***pos***: armazena o valor do registrador na posição de memória rotulada por *pos*.

Com estas instruções disponíveis, podemos escrever códigos de programas simples que poderiam virtualmente ser executados neste nosso computador hipotético. Por exemplo, considere um programa que captura um valor fornecido pelo usuário via teclado e exiba na tela o resultado da soma deste valor com o valor 2.5. Um código que implementa este programa é apresentado a seguir:

```
read 0
storeconst 2.5 1
add 0 1
store 2
write 2
```

Se esse código fosse executado no nosso computador, ao final do programa, o valor entrado pelo usuário estaria armazenado na posição de memória rotulada por 0 e a posição rotulada por 2 teria o valor fornecido acrescido de 2.5. A posição da memória rotulada por 1 foi usada para armazenar a constante 2.5, para que a adição pudesse ser realizada. O valor armazenado em 2 estaria exibido na tela.

Note que este mesmo programa poderia ser re-escrito da seguinte forma:

```
read 0
storeconst 2.5 1
add 0 1
store 0
write 0
```

Neste caso, o resultado exibido na tela seria exatamente o mesmo, mas o programa sobre-escreve o valor armazenado na posição 0. Este programa também seria um programa válido, isto é, podemos sobre-escrever valores armazenados na memória.

Em contrapartida, note que o programa a seguir estaria **errado**:

```
read 0
storeconst 2.5 1
add 0 1
store 0
write 2
```

O erro consiste na tentativa de exibir o valor armazenado na posição 2, que não foi definido. Este programa exibiria na tela um valor indefinido (que usualmente chamamos de "lixo"), que estaria no momento armazenado na posição 2 da memória. De maneira análoga, o programa a seguir também estaria **errado**:

```
read 0
add 0 1
store 2
write 2
```

Neste caso, o erro está na tentativa de somar o valor da posição 0 com o valor da posição 1, uma vez que a posição 1 não teve seu valor previamente definido. O resultado da soma seria então um valor indefinido (um "lixo") que então seria armazenado na posição 2 e exibido na tela.

Estes exemplos, apesar de simples, tratam de dois conceitos importantes que temos que estar cientes quando programamos um computador: devemos especificar uma posição da memória para armazenarmos os valores e devemos definir os valores destas posições antes de usá-los em operações.

1.2 Armazenamento de valores na memória

Assim como num computador real, vamos imaginar que a memória do nosso computador hipotético seja um componente físico e, portanto, limitado. Vamos considerar que cada posição de memória seja subdividida, por exemplo, em oito espaços. Em cada espaço podemos armazenar um algarismo (de 0 a 9). O número zero, por exemplo, seria representado armazenando zeros em todos os espaços:

|0|0|0|0|0|0|0|0|

Se considerarmos apenas a representação de valores inteiros positivos, podemos representar 10^8 valores distintos, e o maior valor inteiro positivo seria:

|9|9|9|9|9|9|9|9|

Uma forma simples de armazenarmos também valores negativos seria reservarmos um dos espaços disponíveis (por exemplo, o primeiro) para indicar se o número é positivo ou negativo. Podemos usar a seguinte convenção: se neste espaço o valor armazenado for zero, o número é positivo; se for um, o número é negativo. Assim, a representação do valor -457 seria:

|1|0|0|0|0|4|5|7|

onde o primeiro algarismo indica tratar-se de um número negativo. Podemos estender a discussão para representação de números reais. Primeiramente, isso pode ser feito reservando-se parte dos espaços para a parte inteira e parte para a parte fracionária, ambas seguindo a representação de números inteiros discutida. Supondo que temos 5 posições para a parte inteira (incluindo o sinal) e 3 posições para a parte fracionária, o número -257.4 seria representado por:

|1|0|2|5|7|4|0|0|

Esta forma de representação é conhecida como representação em *ponto fixo*, pois o ponto decimal está sempre implicitamente representado na mesma posição. Note que, com esta representação, não seria possível representar valores reais iguais ou maiores que 10000 nem valores positivos menores que 0.001, o que seria muito limitante.

Uma outra forma possível para o armazenamento de valores reais é utilizarmos a notação científica. O número -257.4 pode ser escrito da seguinte forma: -0.2574×10^3. Com esta notação, não temos que representar a parte inteira. Podemos então utilizar alguns espaços para representar o expoente da base 10 (com sinal) e o restante para representarmos a mantissa (parte fracionária, com sinal). Considerando cinco espaços para a mantissa e três espaços para o expoente, o mesmo -257.4 seria representado por:

|1|2|5|7|4|0|0|3|

Esta forma de representação é conhecida como representação em *ponto flutuante*, pois a posição do ponto decimal do número varia conforme o valor do expoente. Note que agora podemos representar valores que variam de $\pm 0.9999 \times 10^{99}$ a $\pm 0.0001 \times 10^{-99}$. De qualquer forma, temos uma representação finita. O resultado de uma multiplicação de dois valores armazenados

em memória pode resultar num valor maior do que o máximo valor que pode ser representado na memória. Por exemplo, os valores 0.23×10^{50} e -0.455×10^{60} podem ser representados, respectivamente, por:

```
|0|2|3|0|0|0|5|0|
|1|4|5|5|0|0|6|0|
```

No entanto, se instruímos o computador para multiplicar os dois valores, ocorre um erro, pois o valor resultante (-0.10456×10^{110}) não pode ser armazenado na memória. Em computação este erro é conhecido como *overflow*. Um erro similar é quando o expoente alcança um valor negativo que não pode ser representado (por exemplo, 10^{-110}). Neste caso, o valor é muito próximo de zero mas não pode ser precisamente armazenado na memória. Este erro é conhecido como *underflow*.

Outro possível "erro", mais sutil e muito comum, ocorre quando, por exemplo, o resultado de uma divisão gera um número com mais números significativos do que pode ser representado na memória (seria o caso do número de algarismos significativos ultrapassar o limite de espaços que temos reservado para representar a mantissa).Por exemplo, o resultado da divisão de 23.5 por 8 que resulta no valor 2.9375 seria representado no nosso computador como sendo 2.937 (pois só temos 5 espaços para armazenar a mantissa com sinal):

```
|0|2|9|3|7|0|0|1|
```

Neste caso, perdemos precisão numérica ao armazenar o número na memória. Esse é um erro inerente quando fazemos computações numéricas no computador. Em cálculos numéricos sofisticados, devemos estar cientes da perda de precisão nas operações.

Por fim, note que temos representações diferentes para armazenar valores inteiros e reais. Isso também acontece nos computadores reais. Para otimizar a utilização da memória do computador usa-se representações diferentes de acordo com o tipo de valor a ser armazenado. Assim, se num determinado espaço de memória sabe-se antecipadamente que só serão armazenados valores inteiros, usa-se a representação de inteiros. Se em outra posição pretende-se armazenar valores reais, usa-se a representação de reais.

Exercícios

Usando a representação de ponto flutuante descrita para nosso computador hipotético, indique a representação dos valores de x resultantes das seguintes operações:

1. $x = 1.34 \times 2.3$
2. $x = 20.423 + 90.5$
3. $x = -20.5 \div 3$

1.3 Representação de códigos

Para que nosso computador hipotético seja capaz de executar um programa, é necessário que este programa, isto é, a seqüência de instruções correspondente, esteja armazenado na memória do computador. Como a memória só armazena valores numéricos, temos que criar códigos numéricos para cada possível instrução. Considerando as instruções listadas anteriormente, podemos imaginar a seguinte associação com códigos numéricos:

- **read**: 0
- **write**: 1

- **storeconst**: 2
- **add**: 3
- **sub**: 4
- **mul**: 5
- **div**: 6
- **store:** 7

Com isto, o código:

```
read 0
storeconst 2.5 1
add 0 1
store 2
write 2
```

pode ser representado na memória do nosso computador hipotético da forma abaixo. Usa-se representação de inteiros para armazenar os códigos numéricos das instruções e posição da memória e usa-se representação de reais para armazenar constantes numéricas reais.

```
|0|0|0|0|0|0|0|0|                # código de 'read': 0
|0|0|0|0|0|0|0|0|                # 0
|0|0|0|0|0|0|0|2|                # código de 'storeconst': 2
|0|2|5|0|0|0|0|1|                # 2.5 (valor real)
|0|0|0|0|0|0|0|1|                # 1
|0|0|0|0|0|0|0|3|                # código de 'add': 3
|0|0|0|0|0|0|0|0|                # 0
|0|0|0|0|0|0|0|1|                # 1
|0|0|0|0|0|0|0|7|                # código de 'store': 7
|0|0|0|0|0|0|0|2|                # 2
|0|0|0|0|0|0|0|1|                # código de 'write': 1
|0|0|0|0|0|0|0|2|                # 2
```

Desta forma, o programa pode ser armazenado na memória do computador e então ser executado.

Exercícios

Considerando os códigos numéricos das instruções do nosso computador hipotético, decifre o que faz os códigos a seguir (ambos são programas para calcular áreas de figuras geométricas planas):

1. ```
   |0|0|0|0|0|0|0|0|
   |0|0|0|0|0|0|0|0|
   |0|0|0|0|0|0|0|0|
   |0|0|0|0|0|0|0|1|
   |0|0|0|0|0|0|0|5|
   |0|0|0|0|0|0|0|0|
   |0|0|0|0|0|0|0|1|
   |0|0|0|0|0|0|0|7|
   |0|0|0|0|0|0|0|2|
   |0|0|0|0|0|0|0|1|
   |0|0|0|0|0|0|0|2|
   ```
   ```

2. |0|0|0|0|0|0|0|0|
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	5
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	2
0	3	1	4	2	0	0	1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	5
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	2

1.4 Exemplo de programa simples

Agora que já sabemos como programas e dados podem ser armazenados na memória, vamos praticar um pouco de programação usando este nosso computador hipotético. Nosso primeiro programa terá por objetivo converter valores de uma unidade para outra. Vamos considerar que queremos desenvolver um programa que capture um valor de uma temperatura expressa em graus Celsius (c) e exiba a temperatura em graus Fahrenheit (f) correspondente. Sabemos que a fórmula de conversão entre estas duas unidades de temperatura é expressa por: $f = 1.8c + 32$. Nosso programa pode então ser codificado como:

```
read 0
storeconst 1.8 1
mul 1 0
store 2
storeconst 32 3
add 2 3
store 4
write 4
```

Observe que, além das posições de memória usadas para armazenar o valor fornecido (posição 0) e as constantes 1.8 (posição 1) e 32 (posição 3), foi necessário utilizar um espaço de memória auxiliar (posição 2) para armazenar um valor temporário, necessário para calcular o valor final, armazenado na posição 4. Logicamente, outras soluções também são possíveis.

1.5 Repetições

Com as instruções apresentadas até aqui, nosso computador hipotético fica limitado a programas que aplicam diretamente fórmulas matemáticas. Problemas mais complexos precisam de mais recursos de programação. Muitos problemas, que têm soluções diretas complexas, podem ser facilmente resolvidos de forma incremental, fazendo cálculos simples repetidas vezes. Como computadores são máquinas capazes de realizar cálculos de forma muito eficiente, dividir um problema complexo em diversas computações simples é uma estratégia de programação de computadores muito adequada e muito utilizada.

Um exemplo clássico de problema de programação que pode ser resolvido com repetições é o cálculo do fatorial de um número inteiro não negativo. Por definição, temos:

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1 & \text{se } n > 0. \end{cases}$$

Vamos então pensar em escrever um programa que, dado um valor de n, fornecido pelo usuário, calcule e exiba o valor do fatorial na tela do computador. Como não sabemos *a priori* o valor de n, não há como computar diretamente o valor do fatorial. Se tivermos uma maneira de repetir uma determinada computação, podemos acumular o valor de várias multiplicações, fazendo uma multiplicação por vez.

Para que nosso computador hipotético seja capaz de executar repetidamente uma determinada seqüência de instruções, vamos adicionar uma nova instrução:

- **jump *pos offset***: pula o número de instruções (na seqüência do programa) especificado em *offset* se o valor armazenado na posição *pos* for diferente de zero; caso contrário, continua a execução com a instrução que se segue. Observe que o valor de *offset* pode ser negativo, voltando a execução do programa para alguma instrução anterior.

Com este novo mecanismo, podemos pensar num programa para nosso computador hipotético fazer o cálculo do fatorial de um número fornecido via teclado pelo usuário. Uma possível solução é descrita a seguir. Vamos armazenar o valor de n na posição 0 da memória e armazenar o valor do fatorial calculado na posição 2. Como o valor do fatorial será obtido acumulando o valor de multiplicações, temos que inicializar o valor do fatorial com o valor neutro da multiplicação, isto é, com o valor 1. A seguir, devemos tratar o caso especial: se $n = 0$, o valor armazenado em 2 já é a solução do problema. Se o valor fornecido não for igual a zero, podemos fazer a seguinte construção: multiplicamos o valor em 0 pelo fatorial atual (valor armazenado na posição 2), armazenando o resultado sobre-escrevendo a posição 2. Em seguida, diminuímos o valor em 0 (valor originalmente de n) de uma unidade e repetimos o processo até que o valor da posição 0 chegue a zero. Nesta hora, teremos na posição 2 o valor do fatorial armazenado. A posição 1 da memória é utilizada para armazenar a constante 1, necessária para decrementar o valor fornecido a cada repetição (iteração). O código deste procedimento para nosso computador hipotético é apresentado a seguir:

```
read 0            # captura valor e armazena na posição 0
storeconst 1 1    # armazena na posição 1 o valor constante 1
storeconst 1 2    # armazena o valor inicial do fatorial (1) na posição 2
jump 1 5          # faça o controle avançar para próximo jump (necessário se n=0)
mul 2 0           # multiplica valor em 2 por valor em 0
store 2           # sobre−escreve valor em 2
sub 0 1           # subtrai uma unidade do valor em 0
store 0           # sobre−escreve o valor em 0
jump 0 −4         # repete computação se valor em 0 não for zero
write 2           # escreve o valor do fatorial na tela
```

Exercícios

Usando o computador hipotético descrito neste capítulo:

1. Escreva um programa para converter o valor de uma temperatura dada em graus Fahrenheit para graus Celsius.

2. Escreva um programa que capture o valor do raio de uma esfera e exiba na tela o valor do volume e da área da superfície da esfera. Sabe-se que o volume da esfera é dado por $volume = \frac{4}{3}\pi r^3$ e que a área da superfície é dada por $area = 4\pi r^2$.

3. Simule a execução passo a passo do programa abaixo e mostre a saída gerada na tela do computador.

```
storeconst 0 0
storeconst 1 1
storeconst 10 2
write 0
add 0 1
store 0
sub 2 1
store 2
jump 2 -5
```

4. Escreva um programa que capture dois valores positivos (a e b) fornecidos pelo usuário, com $a \leq b$, e imprima o valor do somatório de a a b, isto é: $a + (a + 1) + \cdots + (b - 1) + b$.

1.6 Computador real

Apesar de simples, nosso computador hipotético ilustra o funcionamento de um computador real. Duas principais características diferem nosso computador hipotético de um computador real: um computador real tem capacidade para armazenar muitos dados na memória e tem capacidade para executar instruções de forma extremamente eficiente.

A representação de valores na memória de um computador real é análoga ao que foi apresentado para nosso computador hipotético. A maior diferença é que nosso computador hipotético trabalhava na base decimal, isto é, em cada espaço da memória podíamos armazenar um valor de 0 a 9. O computador real, por razões tecnológicas, trabalha na base binária, isto é, a menor unidade de espaço de memória só pode armazenar o valor 0 ou 1. A memória de um computador real é dividida em unidades de armazenamento chamadas *bytes*. Cada byte é composto por 8 *bits*. Cada bit pode armazenar o valor *zero* (desligado ou desativado) ou *um* (ligado ou ativado), apenas. Por este motivo, os valores numéricos, para serem armazenados na memória do computador, são representados na base binária. Na representação binária, os números são representados por uma seqüência de zeros e uns. No nosso dia-a-dia (e no nosso computador hipotético), representamos os números na base decimal, uma vez que representamos os números com 10 algarismos, de 0 a 9. Da mesma forma que podemos representar os números no nosso sistema com dez algarismos, podemos também representar números na base binária, isto é, com apenas dois algarismos (0 e 1). Por exemplo, na base decimal, o número 456 representa o valor $4 * 10^2 + 5 * 10^1 + 6 * 10^0$. Isto é, o algarismo da centena (4) é multiplicado pela base (10) elevada ao expoente da centena (2); o algarismo da dezena (5) é multiplicado pela base (10) elevada ao expoente da dezena (1), e assim por diante. De forma análoga, podemos representar um número na base binária. Por exemplo, o número 101 na base binária representa o valor 5 na base decimal, pois $1 * 2^2 + 0 * 2^1 + 1 * 2^0$ é igual a 5.

No sistema decimal, com 8 algarismos podemos representar 100 000 000 valores distintos ($= 10^8$). De forma similar, no sistema binário, com 8 algarismos podemos escrever 256 valores distintos ($= 2^8$). Assim, se usamos um byte para representar um número inteiro não negativo, podemos representar 256 valores distintos, variando de 0 a 255. Em geral, precisamos de mais representatividade e então usamos mais memória (mais bytes) para representar um número. Por exemplo, com 4 bytes, podemos representar 4 294 967 296 valores inteiros distintos ($= 2^{32}$).

Como no nosso computador hipotético, nos computadores reais, cada tipo de valor segue uma regra de representação. Valores de diferentes tipos (por exemplo, inteiro ou real) têm representações internas distintas. A descrição feita se refere a representação de números inteiros não negativos. Valores inteiros negativos e valores reais também são representados na base binária, usando um determinado número de bytes. Detalhes de como estes valores são represen-

tados na base binária fogem do escopo deste texto (o esquema descrito em nosso computador hipotético para armazenar números reais também é aplicado a computadores reais, mas usando a base binária). O importante é entender que os valores numéricos são representados usando um determinado espaço de memória e, portanto, têm representatividade limitada (não é qualquer número que pode ser armazenado no espaço de memória reservado). Quando fazemos cálculos numéricos sofisticados com o computador, temos que estar cientes que as operações são feitas dentro de uma precisão finita, o que pode acarretar em erros de precisão numérica. Um exemplo de erro simples seria tentarmos armazenar valores inteiros positivos maiores que 255 em um único byte. Outro erro de precisão comum é tentarmos armazenar o resultado de uma divisão que resultou em número de casas decimais muito grande (ou mesmo em uma dízima). Neste último caso, o número será truncado de acordo com o espaço de memória que usamos para armazená-lo.

Sabemos que o computador é capaz de processar outros tipos de informação, além de valores numéricos, como textos (mensagens, documentos, etc.). Estas informações são armazenadas em memória, usando um determinado número de bytes. Para que o armazenamento de textos seja possível, cria-se um código numérico para cada caractere. Pode-se, por exemplo, associar o código numérico 65 à letra 'A', o 66 à letra 'B' e assim por diante. Criando-se um código para cada caractere, inclusive os caracteres de pontuação e formatação, pode-se armazenar um texto, isto é, uma seqüência de caracteres, armazenando a seqüência correspondente de códigos numéricos. Para o nosso alfabeto, um byte é suficiente para armazenar os códigos associados aos caracteres, pois não temos mais do que 256 caracteres distintos. Assim, um texto é armazenado numa seqüência de bytes, um para cada caractere do texto.

De forma análoga, podemos armazenar outros tipos de informação no computador. Uma imagem, por exemplo, é constituída por uma seqüência de elementos de imagem, chamados *pixels*. A resolução de uma imagem é definida pela largura e pela altura da imagem, dadas em número de pixels. Numa imagem em tom de cinza, cada pixel armazena um valor numérico que representa a intensidade (variando de preto até branco), sendo armazenado em um byte. Imagens coloridas armazenam três componentes por pixel: a intensidade de vermelho, verde e azul. Assim, cada pixel de uma imagem colorida precisa de três bytes. Quando compramos uma máquina fotográfica digital, nos preocupamos com a quantidade de bytes disponível (por exemplo, 5 MegaBytes que equivale a $5*1024*1024$ bytes). Essa memória é usada para armazenar cada imagem capturada – quanto maior a quantidade de bytes, maior resolução teremos na foto.

Além de informações, como vimos, precisamos também armazenar *programas* na memória do computador. A CPU executa um programa realizando a seqüência de operações especificada pelo programa. Para que a CPU seja capaz de executar as operações é necessário que o programa e os dados acessados estejam na memória. Num computador real, um programa é representado na memória através de uma seqüência de instruções básicas, codificadas também na base binária, de maneira análoga ao que fizemos no nosso computador hipotético: cada instrução recebe um código numérico correspondente.

Na prática, precisamos utilizar muitos recursos para podermos usar o computador para resolver problemas complexos. É fácil imaginar também que mesmo se tivéssemos um conjunto completo de instruções no nosso computador hipotético, a linguagem que usamos para especificar nosso programa é muito rudimentar. Para problemas reais precisamos de uma linguagem de programação de nível mais abstrato. Nos capítulos seguintes, utilizaremos a linguagem de programação C para programar computadores reais.