Capítulo 2: Introdução à Linguagem C

Waldemar Celes e Roberto Ierusalimschy

29 de Fevereiro de 2012

1 Ciclo de desenvolvimento

O modelo hipotético de computador utilizado no capítulo anterior, embora muito simples, serviu para ilustrar o controle do espaço de memória que é necessário para armazenarmos informações na memória do computador. No entanto, com os recursos de programação apresentados, é fácil imaginar que seria impossível fazer a gerência dos espaços de memória em uso num computador real, em especial quando consideramos que existem diversas aplicações sendo executadas simultaneamente, todas usando a memória disponível no sistema. Um computador real também oferece muito mais possibilidades, sendo capaz de processar programas sofisticados. Seria muito difícil programarmos um computador real usando diretamente a linguagem nativa da máquina. Precisamos de uma linguagem de programação que nos permita desenvolver os programas num nível de abstração maior, isto é, uma linguagem mais adequada para que nós, humanos, possamos nos expressar com mais facilidade. Existem diversas linguagens de programação disponíveis: Fortran, Pascal, C, C++, C#, Java, Lua, Python, Scheme, etc. Na prática, a escolha de qual linguagem de programação usar depende da aplicação que queremos desenvolver e do nosso conhecimento das linguagens. Não existe uma linguagem definitiva para todas as aplicações. Da mesma forma, não existe uma linguagem definitiva para ser usada em todos os textos introdutórios de programação.

Neste texto, adotaremos a linguagem de programação C. C é a linguagem básica de programação de maior uso atualmente. Reconhecemos, no entanto, que existem dificuldades no aprendizado de C. Para facilitar este aprendizado, vamos trabalhar com um sub-conjunto dos recursos presentes em C, o suficiente para introduzirmos os principais conceitos de programação. Um ponto adicional a favor da escolha de C é que a aprendizagem de qualquer outra linguagem de programação, incluindo as linguagens orientadas a objetos tais como C++, Java e C#, tende a ser facilitada se programamos em C com desenvoltura.

Logicamente, um programa em C deve respeitar a sintaxe da linguagem. Devemos conhecer a sintaxe de C e saber fazer uso da linguagem para expressarmos as operações que desejamos que o computador realize. Naturalmente, o computador não é capaz de executar uma seqüência de instruções escritas em C (ou em qualquer outra linguagem de alto nível). O computador só é capaz de executar uma seqüência de instruções escritas em sua linguagem nativa, isto é, em sua linguagem de máquina. Portanto, para executar um programa em C, devemos antes converter o programa em C para um programa em linguagem de máquina. Esta conversão chama-se *compilação* e o programa que faz a conversão chama-se *compilador*. Existem diferentes compiladores da linguagem C disponíveis. Qualquer um que siga o padrão internacional da linguagem C (ISO/ANSI C) pode ser usado.

A Figura 1 ilustra o processo de desenvolvimento de programas com a linguagem C. Primeiramente, escrevemos o código do programa usando um editor de texto convencional. Podemos usar qualquer editor de texto, desde que seja um editor que não trabalhe com formatação (não tem sentido definir parágrafos, escrever em negrito, etc.). Vários editores de texto são especializados para programação e automaticamente definem cores distintas para as partes do código, facilitando a identificação da sintaxe da linguagem e o entendimento do código. Uma vez ter-

minada a edição do código, devemos salvar nosso código num arquivo (usando a extensão .c) e passamos para a etapa da compilação. O compilador é um programa que usamos para converter o código escrito em C no código correspondente em linguagem de máquina, resultando num programa que pode ser executado no computador. Durante a compilação, o compilador identifica eventuais erros de sintaxe no código. A compilação só pode ser efetuada se o código em C for livre de erros de sintaxe. Se o compilador reportar erros, devemos re-editar o código em C para corrigí-los e então tentar compilar novamente. Se a compilação for bem sucedida, um programa executável (com extensão .exe, no Windows) é gerado. Podemos então executar nosso programa. Isso não necessariamente termina o ciclo de desenvolvimento, pois devemos verificar se o programa executa de forma esperada, isto é, devemos verificar se o programa está correto (sim, o compilador já verificou a sintaxe do programa, mas o programa pode não estar funcionando da maneira esperada, pois podemos ter cometido erros na elaboração da solução do problema que queremos resolver). Infelizmente, como o computador é meramente uma máquina, um pequeno erro de programação em geral invalida completamente o programa. Após testarmos nosso programa e verificarmos que este funciona da maneira esperada, podemos considerar o desenvolvimento concluído.

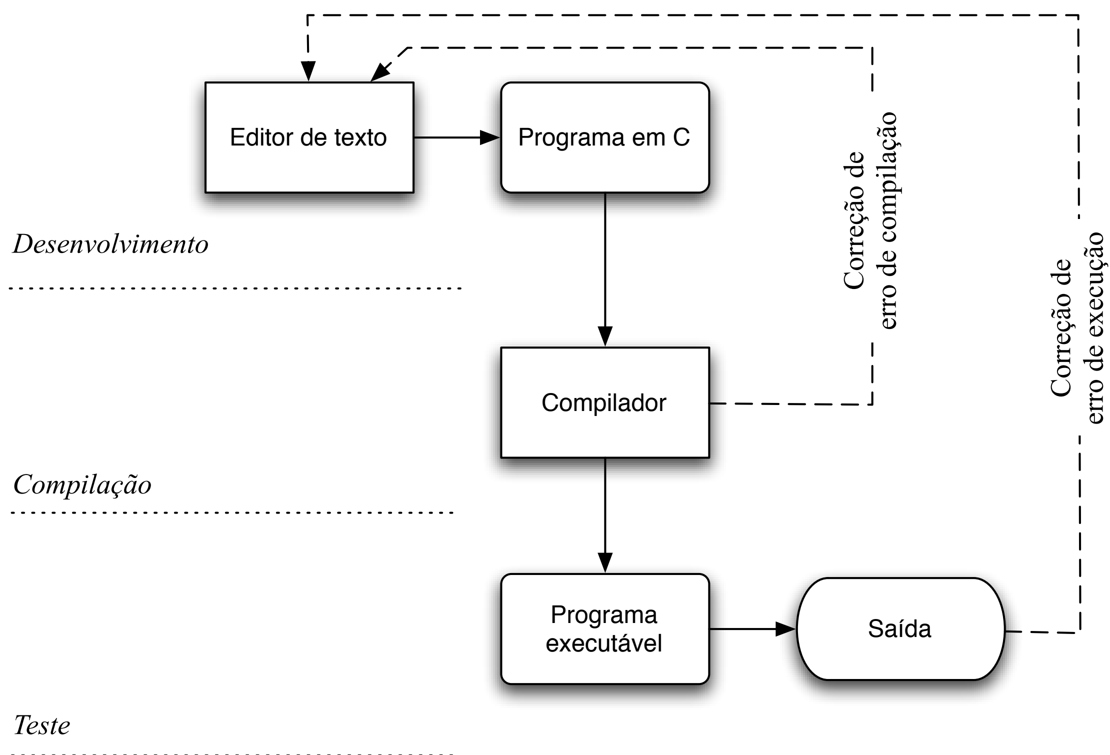

Figura 1: Ciclo de desenvolvimento de um programa.

2 Programando em C

Considerando um computador real e o uso da linguagem C, vamos retomar nosso programa que converte valores de temperatura de graus Celsius para graus Fahrenheit. Relembrando, queremos desenvolver um programa que capture um número real, representando o valor de temperatura em graus Celsius fornecido pelo usuário via teclado, e exiba o valor correspondente, em graus Fahrenheit, na tela. Um código em C que implementa uma solução deste problema é apresentado a seguir. O leitor não deve se preocupar em memorizar todos os conceitos que serão apresentados. Por ora, o importante é conhecer a "cara" de um programa em C. Em seguida, vamos discutir

a solução passo a passo, a fim de introduzir os principais elementos de um programa em C.

```c
#include <stdio.h>

int main (void) {
  float cels;
  float fahr;
  scanf("%f", &cels);
  fahr = 1.8 * cels + 32;
  printf("%f", fahr);
  return 0;
}
```

De maneira análoga ao que fizemos na solução usando nosso computador hipotético do capítulo anterior, temos que determinar espaços de memória para armazenar os valores que usamos para computar a solução do nosso problema. Com a linguagem C, não definimos qual posição da memória física estaremos usando. Apenas requeremos um determinado espaço, escolhendo um nome para referenciá-lo. Ao requerer um espaço de memória, também temos que indicar que tipo de valor será armazenado neste espaço, pois cada tipo de valor (inteiro, real, caractere, etc.) é internamente representado de maneira diferente, usando um determinado número de bytes. Portanto, para requerer um espaço de memória para representar um valor, temos que definir o tipo do valor que será armazenado e escolher um nome para referenciar este espaço de memória.

No nosso programa, precisamos de dois espaços de memória: um para armazenar o valor da temperatura em Celsius e outro para o valor em Fahrenheit. No nosso programa, para requerer estes dois espaços, designados respectivamente por `cels` e por `fahr`, para armazenar números reais, fizemos:

```c
float cels;
float fahr;
```

Note que os comandos em C terminam com um ponto-e-vírgula (`;`).

Para ler um valor real (tipo `float`) fornecido via teclado, podemos usar a função `scanf`, definida na biblioteca padrão chamada `stdio` da linguagem C. Uma *função* em C é um procedimento que quando chamado (invocado) é executado, podendo resultar num valor. A função `scanf`, por exemplo, captura um número real fornecido via teclado e resulta no valor capturado. Portanto, para ter o valor fornecido pelo usuário armazenado no espaço referenciado por `cels` escrevemos:

```c
scanf("%f", &cels);
```

Note que para chamar (invocar) uma função, `scanf` no caso, devemos colocar um abre e fecha parênteses após o nome da função com os parâmetros utilizados pela função (Funções serão tratadas em detalhes no Capítulo 3). Note ainda que quando esta expressão for avaliada, isto é, quando o computador executar este comando, o programa vai esperar o usuário fornecer um valor real via teclado. Quando o usuário fornecer o valor, a chamada da função escreverá o valor fornecido pelo usuário no espaço referenciado por `cels`.

Para calcular o valor da temperatura correspondente em Fahreinheit e armazená-lo no espaço de memória referenciado por `fahr`, usamos a seguinte expressão (`*` representa o operador de multiplicação):

```c
fahr = 1.8 * cels + 32;
```

Na avaliação desta expressão, o computador fará a multiplicação de `1.8` pelo valor que estará armazenado no espaço referenciado por `cels` e o resultado será somado com `32`. O valor final da expressão será então armazenado no espaço referenciado por `fahr`. Note o significado do sinal de igual (`=`): *atribuição*.

Para exibir o resultado armazenado em `fahr` na tela, podemos usar a função `printf`, também definida na biblioteca padrão `stdio` que estamos considerando. Assim, escrevemos:

```
printf("%f", fahr);
```

Note novamente que `printf` é uma função, daí a necessidade dos parênteses após seu nome. Note ainda que estamos passando o valor referenciado por `fahr` para a função, especificando-o entre os parênteses (temos que passar o valor para a função pois é a função que efetivamente exibe o valor na tela).

Esta sequência de comandos segue a sintaxe de C e implementa as operações que devem ser feitas para capturar, calcular e exibir um valor de temperatura. No entanto, um programa em C não é composto simplesmente de uma seqüência de comandos. Os comandos num programa em C devem estar agrupados em *funções*. Escrever uma programa em C é escrever um conjunto de funções. Cada função é composta por uma seqüência de comandos. Nós, programadores, somos responsáveis por definir como podemos construir a solução do nosso problema implementando um determinado conjunto de funções. No entanto, para que um programa em C possa ser executado, é necessário que exista uma função principal, denominada *main*. Assim, um programa em C pode ter várias funções, mas uma delas tem que ser a *main*. Esta é a função onde se inicia a execução do programa. No nosso exemplo, a computação que estamos fazendo é tão simples que podemos não escrever outras função, apenas a *main*. Agrupamos então nossos comandos dentro da função *main*:

```
int main (void)
{
  float cels;
  float fahr;
  scanf("%f", &cels);
  fahr = 1.8 * cels + 32;
  printf("%f", fahr);
  return 0;
}
```

A linha `int main (void)` representa o cabeçalho da função. Como veremos mais adiante, uma função pode receber parâmetros e pode retornar um valor. No caso da função *main*, como trata-se de uma função especial, ela não recebe parâmetros (por isso o `void` após os parênteses) e tem que retornar um valor inteiro (por isso o `int` antes do nome da função)[1]. O *corpo* da função é delimitado por chaves (`{ ... }`) e contém os comandos que compõem a função. A última linha da função (`return 0;`) é necessária pois a função deve retornar um valor inteiro (e, em geral, retornamos o valor zero na função *main*).

Para o nosso programa ficar completo, precisamos apenas incluir a *interface* de eventuais bibliotecas usadas. No nosso caso, estamos usando funções de uma biblioteca padrão para as operações de entrada (captura de valores via teclado) e saída (exibição de valores na tela). O nome desta biblioteca padrão é *stdio* (*standard input/output library*). Acrescentamos então uma linha ao nosso código, antes da definição da função *main*:

```
#include <stdio.h>
```

[1]A função *main* pode receber parâmetros, mas isto não será tratado neste texto. O valor de retorno da função deve ser sempre um inteiro.

Isto completa as linhas do programa apresentado. Podemos compilar e executar o programa para verificar se funciona como esperado.

Podemos ainda fazer uma melhoria no nosso programa. Quando executarmos o programa (logicamente após a compilação), logo no início a execução será interrompida para esperar que o usuário entre com um valor de temperatura (chamada da função `scanf`). Se passarmos este programa para um outro usuário usar, este usuário não saberá que deve fornecer o valor de uma temperatura. Para orientar o usuário podemos exibir uma mensagem na tela antes da chamada da função `scanf`. Para isso, podemos usar a função `printf`). Por exemplo, podemos incluir o seguinte comando:

```
printf("Entre com temperatura em Celsius: ");
```

Podemos melhorar também a saída do nosso programa, incluindo uma mensagem explicando o significado do valor exibido na tela. Finalmente, para concluir, podemos acrescentar comentários no nosso código. Os comentários em C são delimitados por `/* ... */`. Comentários não alteram o código (os compiladores desprezam os comentários), mas eles podem ser úteis para documentar o código. A versão do programa com estas melhorias é apresentada a seguir:

```
/* Programa para converter temperatura de Celsius para Fahrenheit */

#include <stdio.h>   /* biblioteca padrão de entrada e saída */

int main (void)
{
   float cels;   /* espaço para armazenar temperatura em Celsius */
   float fahr;   /* espaço para armazenar temperatura em Fahrenheit */

   /* captura valor fornecido via teclado */
   printf("Entre com temperatura em Celsius: ");
   scanf("%f", &cels);

   /* faz a conversão */
   fahr = 1.8 * cels + 32;

   /* exibe resultado na tela */
   printf("Temperatura em Fahrenheit: ");
   printf("%f", fahr);

   return 0;
}
```

A linguagem C não impõe nenhuma regra de formatação por linha na codificação de programas. Podemos, por exemplo, escrever todos os comandos do código numa única linha. Logicamente, seria muito difícil entender um código escrito desta forma. Em geral, buscamos codificar um comando por linha, mas se o comando for muito longo podemos livremente codificar o comando usando diversas linhas. A existência de linhas em branco no código também é opcional. Para o compilador, a mudança de linha é tratada como um espaço em branco. Em resumo, não existe regra de formatação em C, mas sempre existe o bom senso e práticas comuns de programação. Por exemplo, as linhas de comando que compõem o código da função são deslocadas para a direita, acrescentando espaços em branco no início da linha. Desta forma, fica mais fácil identificar quais comandos compõem a função. A existência destes espaços em branco no início da linha é chamada *indentação*.

3 Variáveis

Como vimos, para armazenarmos valores na memória do computador, precisamos requerer um espaço de memória. Em C, isto é feito através de *declaração de variáveis.* No exemplo para conversão de temperatura, declaramos duas variáveis:

```
float cels;
float fahr;
```

Dizemos que uma *variável* representa um espaço de memória onde podemos armazenar valores de um determinado tipo. Assim, quando declaramos uma variável definimos seu *tipo* (`float`, no exemplo acima) e escolhemos seu *nome* (`cels` e `fahr`, no exemplo). Uma variável é definida pelo seu tipo e seu nome.

O nome de uma variável serve como referência para o espaço de memória associado. Assim, o trecho de código a seguir:

```
float a;
a = 2.5;
```

declara uma variável de tipo `float` e nome `a` e, em seguida, atribui o valor `2.5` à variável, isto é, armazena o valor no espaço de memória associado à variável. Escolhemos livremente os nomes das nossas variáveis. Em geral, procuramos escolher nomes apropriados ao significado da informação armazenada, pois isto facilita nosso entendimento do código. Assim, se queremos armazenar um valor que representa o raio de uma esfera, por exemplo, podemos declarar:

```
float raio;
```

Logicamente, outros nomes também são possíveis: `r`, `r0`, `R`, `Raio`, `RAIO`, `raio_da_esfera`, `RaioDaEsfera`, etc. O nome de uma variável pode conter qualquer combinação de letras, dígitos e *underscore*. A única restrição é que o primeiro caractere não pode ser um dígito.

O tipo da variável define o tipo do dado que pode ser armazenado no espaço reservado. A linguagem C oferece os seguintes principais tipos básicos para o armazenamento de valores numéricos:

int	para armazenar números inteiros.
float	para armazenar números reais (ponto flutuante) de simples precisão.
double	para armazenar números reais de dupla precisão.

A diferença entre o tipo `float` e o tipo `double` é a precisão. O tipo `float` ocupa 4 bytes enquanto o tipo `double` ocupa 8 bytes, tendo portanto mais representatividade. Quando estamos trabalhando com computações numéricas onde a precisão é importante (em simulações físicas, por exemplo), o tipo `double` é necessário. Para computações simples, o tipo `float` é suficiente.

Se declaramos uma variável como sendo do tipo `int`, logicamente só podemos armazenar valores inteiros no espaço de memória associado. Assim, se fizermos:

```
int a;
a = 2.6;
```

o valor armazenado em a é 2 (parte inteira do número 2.6). Não necessariamente isto é um erro de programação, mas os compiladores podem reportar uma advertência quando tentamos armazenar valores reais em variáveis inteiras (o código é gerado de qualquer forma). Um problema similar acontece quando tentamos armazenar um valor de dupla precisão (`double`) numa

variável do tipo `float`. Neste caso, o mesmo valor real será armazenado, mas podemos perder precisão na atribuição.

Por fim, é importante mencionar que podemos, na declaração da variável, inicializar o conteúdo da memória com um determinado valor, isto é, a variável é criada com um valor inicial já definido. Isso se chama *inicialização de variáveis*. Como vimos com o computador hipotético, só podemos usar uma variável numa expressão se esta já tiver seu valor definido (caso contrário a computação resulta num valor indefinido, "lixo"). Por isso, se já soubermos o valor que vamos atribuir à variável, é uma boa prática de programação inicializar as variáveis. O trecho de código a seguir ilustra inicialização de variáveis. Note ainda que podemos declarar e inicializar mais de uma variável do mesmo tipo no mesmo comando:

```
int a = 2;                /* declara e inicializa uma variável */
int b, c = 5;             /* declara duas variáveis, e inicializa uma */
float d = 3.1;            /* declara e inicializa um variável */
float e = 4.5, f = 5.3;   /* declara e inicializa duas variáveis */
```

4 Expressões e operadores

Uma expressão é uma combinação de variáveis, constantes e operadores que, quando avaliada, resulta num valor. No nosso exemplo de conversão de temperatura, usamos a expressão `1.8 * c + 32` para computar o valor da temperatura em Fahrenheit. Quando o computador avalia esta expressão, o valor constante `1.8` é multiplicado pelo valor da variável `c`, e o resultado da multiplicação é adicionado ao valor constante `32`, resultando no valor da expressão.

A linguagem C oferece um rico conjunto de operadores. Nesta seção vamos apresentar apenas os principais operadores aritméticos. Os demais operadores serão apresentados ao longo do texto. Os principais operadores aritméticos são: *adição* (`+`), *subtração* (`-`), *multiplicação* (`*`) e *divisão* (`/`). Todos estes são operadores binários (atuam em dois operandos). Existe ainda o operador *menos unário* (como em: `-2 * 8`). A ordem de avaliação das operações aritméticas numa expressão segue o convencional: multiplicação e divisão têm precedência em relação a adição e subtração. Assim, a expressão `2 + 3 * 4` resulta no valor 14. Se, neste caso, for necessário avaliar a adição antes, podemos usar parânteses: a expressão `(2 + 3) * 4` resulta no valor 20.

Para ilustrar o uso de operadores aritméticos, vamos considerar um programa para calcular o volume de uma esfera de raio fornecido pelo usuário. O programa deve capturar o valor do raio fornecido via teclado e exibir na tela o valor do volume correspondente da esfera.

A solução deste problema é simples, pois precisamos apenas aplicar diretamente a fórmula para o cálculo do volume de uma esfera de raio r, que é dada por: $volume = \frac{4}{3}\pi r^3$. Como a linguagem C não tem o operador de exponenciação (ou potenciação), podemos escrever a expressão multiplicando o raio por ele mesmo três vezes. Outra alternativa seria usar a função de exponenciação (`pow`) presente na biblioteca matemática padrão da linguagem C (se optarmos por usar essa função, teremos que incluir a interface da biblioteca de matemática).

Vamos inicialmente implementar uma solução sem o uso da biblioteca padrão. Uma possível solução do problema é dada a seguir:

```
#include <stdio.h>

int main (void)
{
   float raio;         /* raio da esfera */
   float vol;          /* volume calculado */
```

```
    printf("Entre com o raio da esfera: ");
    scanf("%f", &raio);
    vol = 4.0 / 3.0 * 3.14159 * raio * raio * raio;
    printf("Volume da esfera: ");
    printf("%f", vol);

    return 0;
}
```

Para evitar o uso da constante numérica que representa o valor de π dentro do corpo da função, podemos definir uma *constante simbólica*, deixando o código mais legível, como ilustrado abaixo:

```
#include <stdio.h>

#define PI 3.14159

int main (void)
{
    float raio;         /* raio da esfera */
    float vol;          /* volume calculado */

    printf("Entre com o raio da esfera: ");
    scanf("%f", &raio);
    vol = 4.0 / 3.0 * PI * raio * raio * raio;
    printf("Volume da esfera: ");
    printf("%f", vol);

    return 0;
}
```

Na compilação deste código, todas as ocorrências da palavra `PI` são substituídas pelo texto `3.14159`. Portanto, o compilador processa o mesmo código que antes. A vantagem de usar constantes simbólicas é clareza e facilidade de manutenção. Podemos usar a constante simbólica `PI` em diversas partes do nosso código. Se quisermos mudar a precisão do valor π, por exemplo expressando o valor com mais casas decimais, precisamos mudar apenas a linha da definição.

A versão desta solução usando a função `pow` da biblioteca matemática padrão da linguagem C é apresentada a seguir. A função `pow` recebe dois parâmetros, x e y, e tem como valor de retorno o valor x^y. A interface da biblioteca matemática é definida no arquivo `math.h`. Como trata-se de uma interface da biblioteca padrão, esta é incluída no nosso programa pelo comando: `#include <math.h>`.

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159

int main (void)
{
    float raio;         /* raio da esfera */
    float vol;          /* volume calculado */

    printf("Entre com o raio da esfera: ");
    scanf("%f", &raio);
    vol = 4.0 / 3.0 * PI * pow(raio, 3);
    printf("Volume da esfera: ");
```

```
    printf("%f", vol);

    return 0;
}
```

4.1 Aritmética inteira

Existe um detalhe sutil no exemplo acima que temos que estar sempre atentos quando codificamos expressões aritméticas. Se tivéssemos escrito a expressão para o cálculo do volume assim:

```
vol = 4 / 3 * PI * pow(raio, 3);
```

o resultado obtido não estaria correto. A causa do erro estaria na representação usada na divisão. Em C, uma operação aritmética (em especial, a divisão) é feita na representação dos operandos. Se os dois operandos são números inteiros, a operação é feita na representação inteira. Isto significa que o resultado da operação também é um número inteiro. Assim, a expressão `4 / 3` resulta no número `1` (inteiro) e não no valor `1.3333...` como poderia ser esperado. Vamos considerar o exemplo ilustrado no trecho de código abaixo:

```
int a = 5;
int b = 2;
float c = a/b;
```

Se este trecho de código fosse executado, ao final a variável `c` teria o valor `2.0` (float) armazenado. Isto porque a divisão, envolvendo dois operandos do tipo inteiro (variáveis `a` e `b`), seria feita em representação inteira, resultando no valor inteiro `2`. Em seguida, este valor inteiro seria atribuído à variável `c` (que armazena o valor real correspondente).

Esta característica, em geral, não impõe nenhuma dificuldade extra na programação, basta estarmos atentos. Na verdade, a existência de operações feitas em representação inteira não é um problema e sim uma característica importante que podemos explorar: muitas aplicações tiram proveito das operações em representação inteira. Para ilustrar, vamos considerar um exemplo. Considere uma aplicação que leia do teclado um número inteiro que representa o tempo total em segundos decorrido em uma maratona. Queremos então exibir o tempo correspondente no formato *hora minuto segundo*.

O desenvolvimento desta aplicação é facilitado se trabalhamos com representação inteira. Supondo que o número total de segundos é um número inteiro (`tot`), podemos obter o número de horas fazendo a divisão, em representação inteira, do número total de segundos pela quantidade de segundos existentes em uma hora (`tot / 3600`). Para obter a quantidade de minutos que sobrou, basta fazer a divisão pelo número de segundos em um minuto. Nosso programa pode ser codificado como mostrado a seguir:

```
#include <stdio.h>

int main (void)
{
    int tot;              /* número total de segundos */
    int hor, min, seg;    /* número de horas, minutos e segundos */

    printf("Entre com o numero de segundos transcorridos: ");
    scanf("%d", &tot);

    hor = tot / 3600;
```

```
    tot = tot - (hor * 3600);     /* número de segundos restantes */
    min = tot / 60;
    seg = tot - (min * 60);

    printf("Tempo transcorrido: ");
    printf("%d ", hor);
    printf("%d ", min);
    printf("%d", seg);

    return 0;
}
```

Aplicações que trabalham com representação inteira são tão comuns que a linguagem C oferece um operador aritmético adicional para operandos inteiros. Trata-se do *operador módulo* (%) que recebe dois operandos inteiros e resulta no valor do resto da divisão, em representação inteira, do primeiro operando pelo segundo. Assim, a expressão `8 % 3` resulta no valor 2, pois o resto da divisão de 8 por 3 é 2. Como veremos, existem várias situações em que faremos uso do operador módulo em nossos programas. Um exemplo simples é quando queremos determinar se o valor armazenado em uma dada variável do tipo inteiro (`x`) é par ou ímpar. Para tanto, basta verificar o valor resultante de `x % 2`: se for 0, o número é par; se for 1, o número é ímpar.

Como um exemplo adicional, podemos re-escrever a conversão para hora, minuto e segundo usando o operador módulo (esta solução não necessariamente é mais adequada, serve apenas como exemplo):

```
#include <stdio.h>

int main (void)
{
    int tot;                 /* número total de segundos */
    int hor, min, srg;       /* número de horas, minutos e segundos */

    printf("Entre com o numero de segundos transcorridos: ");
    scanf("%d", &tot);

    hor = tot / 3600;
    min = (tot % 3600) / 60;
    seg = tot % 60;

    printf("Tempo transcorrido: ");
    printf("%d ", hor);
    printf("%d ", min);
    printf("%d", seg);

    return 0;
}
```

4.2 Aritmética mista

Se, numa expressão aritmética, os operandos forem de tipos diferentes, a operação é feita na representação do tipo de maior expressividade. Assim, tanto a expressão `4.0 / 3` como a expressão `4 / 3.0` são avaliadas na representação real, resultando em um número real. Se temos, como no trecho de código mostrado anteriormente, duas variáveis inteiras e queremos que a divisão seja feita em representação real, podemos usar um *operador de conversão de tipo*.

```
int a = 5;
int b = 2;
float c = (float)a / b;
```

Neste caso, o operador `(float)` converte o valor de `a` no real correspondente. Note que a variável `a` continua sendo do tipo inteiro e o valor armazenado nela continua sendo do tipo inteiro. A expressão `(float)a`, quando avaliada, resulta no número real correspondente. Este número então torna-se o numerador da divisão, e a divisão é feita em representação real, sem alterar o valor da variável `a`.

O operador de conversão de tipo pode ser útil em outras situações. O trecho de código a seguir:

```
int a;
float b = 2.6;
a = b;
```

armazena o valor `2` (inteiro) em a. Como dissemos, o compilador pode gerar uma mensagem de advertência (pois isso poderia ser uma desatenção do programador). Se estamos cientes da perda de precisão ao armazenar o número, podemos evitar a mensagem de advertência do compilador convertendo explicitamente o tipo do valor, antes da atribuição (novamente, salientamos que o valor da variável `b` não se altera):

```
int a;
float b = 2.6;
a = (int) b;
```

Exercícios

1. Faça um programa que calcule o preço da gasolina por litro no Brasil se adotássemos o mesmo preço cobrado nos Estados Unidos. O programa deve capturar dois valores fornecidos via teclado: o preço do galão de gasolina praticado nos Estados Unidos (em dólares) e a taxa de conversão do dólar para o real. O programa então deve exibir o preço do litro de gasolina correspondente em reais. Sabe-se que um galão tem 3.7854 litros.

2. Faça um programa que converta um valor de altura dado em metros para o valor correspondente expresso em pés e polegadas. O programa deve capturar o valor em metros fornecido via teclado e exibir na tela a mesma altura expressa em pés e polegadas. Por exemplo, se for fornecido o valor 1.8 (metros), o programa deve exibir os valores 5 (pés) e 10.866 (polegadas). Sabe-se que 1 pé tem 30.48 centímetros e que 1 polegada tem 2.54 centímetros.

3. Faça um programa que converta um valor de ângulo dado em radianos para o valor correspondente expresso em *graus*, *minutos* e *segundos*. Sabe-se que 1 radiano equivale a 57.29578 graus.

4. Considerando a existência de notas (cédulas) nos valores R$ 100, R$ 50, R$ 20, R$ 10, R$ 5, R$ 2 e R$ 1, escreva um programa que capture um valor inteiro em reais (R$) e determine o menor número de notas para se obter o montante fornecido. O programa deve exibir o número de notas para cada um dos valores de nota existentes.