# Capítulo 3: Programando com Funções

Waldemar Celes e Roberto Ierusalimschy

29 de Fevereiro de 2012

## 1 Organização de códigos

Um programa de computador representa a implementação de uma solução de um determinado problema. O processo de desenvolvimento de programas de computador envolve diversas etapas. Logicamente, a primeira etapa é entender o problema que se deseja resolver. Em seguida, podemos fazer uma análise do problema na tentativa de identificar possíveis soluções. Por exemplo, podemos considerar soluções feitas para outros problemas que possam ser adaptadas para o problema em questão. Após a identificação de possíveis soluções, podemos desenvolver o projeto do *software* em si. Devemos pensar em como organizaremos nosso programa, que recursos iremos usar, que possíveis códigos já implementados podemos utilizar ou adaptar etc. Por fim, implementamos nosso projeto, codificando a solução usando uma linguagem de programação.

Na prática, estas etapas não são tão bem definidas. Trata-se de um processo dinâmico onde a realização de uma etapa influencia as outras. Muitas vezes, por exemplo, mudamos o problema a medida que construimos uma solução. Isto pode acontecer por diversas razões. O problema original pode, por exemplo, não ter solução, ou o tempo de computação da solução existente pode ser inaceitável. Por outro lado, a solução encontrada para um determinado problema pode ser facilmente estendida para atender outras situações não previstas no problema original. Por se tratar de um processo dinâmico, estamos constantemente revendo códigos já escritos, seja para corrigir eventuais erros encontrados, seja para usar como modelo para resolver um outro problema. Por isso, é fundamental que nosso programa seja escrito de forma organizada, a fim de facilitar a manutenção, o re-uso, a adaptação do código, durante o processo de desenvolvimento ou no futuro.

Neste capítulo, iremos discutir como programar com *funções*. Funções em C são procedimentos que podem ser executados por outros códigos (outras funções). Em C, as funções representam o mecanismo pelo qual podemos *estender* a linguagem. Nos códigos deste texto, por exemplo, utilizamos funções auxiliares para capturar valores fornecidos pelo usuário via teclado e funções auxiliares para exibir informações na tela. De certa forma, estas funções previamente codificadas representam uma extensão da linguagem. Os códigos que escrevemos nos exemplos aqui mostrados usam estas funções, como se elas fizessem parte da própria linguagem. É através deste tipo de mecanismo de extensão que programadores usam a linguagem C para diversas finalidades. Por exemplo, podemos traçar gráficos na tela do computador se tivermos a nossa disposição uma biblioteca que implementa funções para fazer o traçado dos gráficos. De forma análoga, podemos criar sofisticadas interfaces com menus, botões e listas, desde que tenhamos uma biblioteca com funções para esta finalidade. Em geral, podemos usar a linguagem C para fazer qualquer programa, mas muitas vezes será necessário ter disponível uma biblioteca que estende a linguagem, oferecendo funções relacionadas com o programa que queremos desenvolver.

## 2 Criação de novas funções

Conforme mencionado no capítulo anterior, um programa em C é dividido em pequenas funções. Bons programas em geral são compostos por diversas pequenas funções. Como agrupar o código

em funções é um dos desafios que devemos enfrentar. Como o próprio nome diz, uma função implementa uma funcionalidade. A divisão de um programa em funções ajuda na obtenção de códigos estruturados. Fica mais fácil entender o código, mais fácil manter, mais fácil atualizar e mais fácil re-usar.

Em C, a codificação de uma função segue a sintaxe mostrada abaixo:

```
_tipo_do_retorno_ _nome_da_função_ ( _lista_de_parâmetros_da_função_ )
{
   _corpo_da_função_
   ...
}
```

Para cada função que criamos, escolhemos um *nome*. O nome identifica a função e um programa em C não pode ter duas funções com o mesmo nome. Uma função pode receber dados de entrada, que são os *parâmetros da função*, especificados entre os parênteses que seguem o nome da função. Se uma função não recebe parâmetros, colocamos a palavra `void` entre os parênteses. Uma função também pode ter um *valor de retorno* associado. Antes do nome da função, identificamos o tipo do valor de retorno. Se a função não tem valor de retorno associado, usamos a palavra `void`. Entre o abre e fecha chaves, codificamos o *corpo da função*, que consiste no bloco de comandos que compõem a função.

Para ilustrar, vamos considerar novamente o programa que converte uma temperatura dada em graus Celsius para graus Fahrenheit. No capítulo anterior, implementamos este programa codificando tudo dentro da função *main*. Para este exemplo, é fácil identificar uma solução mais estruturada. Podemos dividir nosso programa em duas funções. Uma primeira função fica responsável por fazer a conversão de unidades, enquanto a função *main* fica responsável por capturar o valor fornecido pelo usuário, chamar a função auxiliar que faz a conversão e exibir o valor convertido na tela. Desta forma, identificamos uma funcionalidade bem definida – conversão de Celsius para Fahrenheit – e a codificamos em uma função em separado. Um código que implementa uma função que faz esta conversão é ilustrado a seguir:

```
float celsius_fahrenheit (float tc)
{
   float tf;
   tf = 1.8 * tc + 32;
   return tf;
}
```

Uma grande vantagem de dividir o programa em funções é aumentar seu potencial de re-uso. No caso, após certificarmos que a função está correta, podemos usar esta mesma função em qualquer outro programa que precise converter temperatura em graus Celsius para Fahrenheit.

Com o auxílio desta função, o código da função *main* fica mais simples, facilitando seu entendimento, pois o detalhe de como a conversão é feita está codificado dentro da função auxiliar.

```
int main (void)
{
   float cels;
   float fahr;

   printf("Entre com temperatura em Celsius: ");
   scanf("%f", &cels);

   fahr = celsius_fahrenheit(cels);
```

```
    printf("Temperatura em Fahrenheit: %f", fahr);

    return 0;
}
```

A conversão de graus Celsius para Fahrenheit é muito simples, mas é fácil imaginar que esta função *main* não seria alterada se a conversão estivesse baseada em computações sofisticadas, pois na função *main* tudo o que nos interessa é o valor resultante da chamada da função.

O programa completo é mostrado a seguir:

```
#include <stdio.h>

float celsius_fahrenheit (float tc)
{
    float tf;                                          /* 06 */
    tf = 1.8 * tc + 32;                                /* 07 */
    return tf;                                         /* 08 */
}

int main (void)
{
    float cels;                                        /* 01 */
    float fahr;                                        /* 02 */

    printf("Entre com temperatura em Celsius: ");      /* 03 */
    scanf("%f", &cels);                                /* 04 */

    fahr = celsius_fahrenheit(cels);                   /* 05 */

    printf("Temperatura em Fahrenheit: %f", fahr);     /* 09 */

    return 0;                                          /* 10 */
}
```

No código, as funções devem ser escritas antes de serem usadas. Assim, como a função auxiliar é usada (invocada) pela função *main*, ela deve aparecer antes no código[1]. É importante saber que a execução do programa sempre se inicia com o código da função *main*, independente da ordem em que escrevemos as funções no nosso código. Os números acrescidos à direita do código, em forma de comentários de C, ilustram a ordem de avaliação dos comandos quando o programa é executado. Quando invocamos uma função, como no comando `fahr = celsius_fahrenheit(cels);`, a CPU passa a executar os comandos da *função chamada* (`celsius_fahrenheit`, no caso). Quando a função chamada retorna, a CPU prossegue a execução dos comandos da *função que chama* (`main`, no caso). Isso também ocorre quando se chama uma função de uma biblioteca externa.

## 3 Parâmetros e valor de retorno

Uma função deve ter sua interface bem definida, tanto do ponto de vista semântico como do ponto de vista sintático. Do ponto de vista semântico, quando projetamos uma função, identificamos sua funcionalidade e com isso definimos que dados de entrada são recebidos e qual o resultado (dado de saída) é produzido pela função. Do ponto de vista sintático, os tipos dos dados de entrada e saída são especificados no cabeçalho da função. Uma função pode receber zero

[1] A rigor, esta restrição pode ser contornada com o uso de *protótipos*, mas vamos deixar esta discussão para depois

ou mais valores de entrada, chamados *parâmetros da função*, e pode resultar em zero ou um valor, chamado *valor de retorno da função*. Os parâmetros de uma função são listados entre os parênteses que seguem o nome da função. Assim, a função `celsius_fahrenheit` do programa anterior, cujo protótipo foi definido por:

```
float celsius_fahrenheit (float tc);
```

só recebe um único parâmetro do tipo `float`, ao qual foi associado o nome `tc`. Esta função tem como retorno um valor do tipo `float`, indicado pelo tipo que precede o nome da função. Note o cabeçalho da função *main*. *Main* é uma função que tem como valor de retorno um inteiro e não recebe parâmetros, indicado pela palavra `void` entre os parênteses: `int main (void)`. Existem ainda funções que recebem parâmetros mas não têm valor de retorno associado. Por exemplo, vamos supor uma função responsável apenas pela impressão de uma temperatura qualquer (no nosso caso, vamos utilizá-la para imprimir a nova temperatura dada em Fahrenheit). Esta função, que chamamos de `print_temp` recebe como parâmetro a temperatura que será exibida, mas não retorna um valor para quem a chamou. O protótipo desta função é dado por:

```
void print_temp (float tf);
```

Como pode ser visto pelo programa completo listado abaixo, a nossa função `print_temp` foi implementada para exibir o valor de uma nova temperatura utilizando apenas 2 casas decimais. O programa completo ficaria assim:

```
#include <stdio.h>

void print_temp(float tf)
{
  printf("%.2f", tf);                                    /* 11 */
  return;                                                /* 12 */
}

float celsius_fahrenheit (float tc)
{
   float tf;                                             /* 06 */
   tf = 1.8 * tc + 32;                                   /* 07 */
   return tf;                                            /* 08 */
}

int main (void)
{
   float cels;                                           /* 01 */
   float fahr;                                           /* 02 */

   printf("Entre com temperatura em Celsius: ");         /* 03 */
   scanf("%f", &cels);                                   /* 04 */

   fahr = celsius_fahrenheit(cels);                      /* 05 */

   printf("Temperatura em Fahrenheit: ");                /* 09 */
   print_temp(fahr);                                     /* 10 */

   return 0;                                             /* 13 */
}
```

Outras função recebem mais do que um parâmetro. Para ilustrar, considere uma função que calcule o volume de um cilindro de raio $r$ e altura $h$. Sabe-se que o volume de um cilindro é dado por $\pi r^2 h$. Uma função para calcular o volume de um cilindro deve receber os valores do raio e da altura como parâmetros e ter como retorno o valor do volume calculado. Uma possível implementação desta função é mostrada a seguir:

```
#define PI  3.14159

float volume_cilindro (float r, float h)
{
   float v;
   v = PI * r * r * h;
   return v;
}
```

Para testar esta função, podemos escrever uma função *main*. Nesta função, os valores do raio e da altura são capturados do teclado e o volume computado é exibido na tela. Esta função *main* é muito similar à função *main* do exemplo de conversão de temperaturas: captura-se valores fornecidos via teclado, invoca-se a função que fará a computação desejada e, então, exibe-se os resultados.

```
int main (void)
{
   float raio, altura, volume;
   printf("Entre com o valor do raio: ");
   scanf("%f", &raio);
   printf("Entre com o valor da altura: ");
   scanf("%f", &altura);

   volume = volume_cilindro(raio,altura);

   printf("Volume do cilindro = %f", volume);
   return 0;
}
```

Note que os valores passados na chamada da função devem corresponder aos parâmetros em número e tipo. No caso, a função `volume_cilindro` espera receber dois parâmetros do tipo `float`. A chamada da função passa dois valores para a função: os valores armazenados nas variáveis `raio` e `altura`, ambas do tipo `float`. O valor retornado pela função é armazenado na variável `volume`, também do tipo `float`.

Note ainda que a chamada de uma função que tem um valor de retorno associado é uma expressão que, quando avaliada, resulta no valor retornado. Assim, uma chamada de função pode aparecer dentro de uma expressão maior. Por exemplo, se quiséssemos calcular o volume de metade do cilindro, poderíamos ter escrito:

```
volume = volume_cilindro(raio,altura) / 2.0;
```

Por fim, é importante notar que, na chamada da função, passa-se *valores* para a função chamada. Assim, qualquer expressão pode ser usada, desde que resulte num valor do tipo esperado. Por exemplo, é válido a chamada de função abaixo, que pede para calcular o volume do cilindro com altura dobrada em relação ao valor entrado pelo usuário:

```
volume = volume_cilindro(raio,2*altura);
```

# 4 Escopo de variáveis

Conforme discutido, requisitamos espaços de memória para o armazenamento de valores através da declaração de variáveis. Quando, durante a execução do código, é encontrada uma declaração de variável, o sistema reserva o espaço de memória correspondente. Este espaço de memória permanece disponível para o programa durante o *tempo de vida* da variável. Uma variável declarada dentro de uma função é chamada *variável local*. Uma variável local tem seu tempo de vida definido pela função que a declarou: a variável existe durante a execução da função. Assim que a função retorna, os espaços de memória reservados para suas variáveis locais são liberados para outros usos, e o programa não pode mais acessar estes espaços. Por esse motivo, as variáveis locais são também classificadas como *variáveis automáticas*. Note que uma função pode ser chamada diversas vezes. Para cada execução da função, os espaços das variáveis locais são automaticamente reservados, sendo liberados ao final da execução.

As variáveis locais são "visíveis" (i.e., podem ser acessadas) apenas após sua declaração e dentro da função em que foi definida. Dizemos que o *escopo* da variável local é a função que a define. Dentro de uma função não se tem acesso a variáveis locais definidas em outras funções. Os *parâmetros de uma função* também são variáveis automáticas com escopo dentro da função. Essas variáveis são inicializadas com os valores passados na chamada da função. Como as variáveis locais, os parâmetros representam variáveis que vivem enquanto a função esta sendo executada.

É importante frisar que as variáveis locais têm escopo dentro da função que as declaram. Para esclarecer, vamos re-escrever a função auxiliar, alterando os nomes das variáveis automáticas (parâmetros e variável local).

```
#define PI  3.14159

float volume_cilindro (float raio, float altura)
{
   float volume;
   volume = PI * raio * raio * altura;
   return volume;
}
```

Note que agora as variáveis da função auxiliar têm os mesmos nomes que as variáveis da função *main*. Isto, no entanto, em nada altera o funcionamento do programa. Apesar dos nomes iguais, são variáveis distintas, alocadas em diferentes áreas de memória. Cada variável está dentro do contexto da função que a declara.

Existem outros tipos de variáveis na linguagem C. Uma variável pode ser declarada fora das funções, no espaço denominado *global*. Uma variável global vive ao longo de toda a execução do programa e é visível por todas as funções subseqüentes. Dentro de uma função, uma variável também pode ser definida como *estática*. Uma variável estática também existe durante toda a execução do programa mas só é visível dentro da função que a declara. Por ora, não trabalharemos com variáveis globais e estáticas.

## 4.1 Modelo de pilha

Esta seção explica os detalhes de como os valores são passados para uma função. Esta explicação pode ajudar o leitor a entender melhor a alocação de variáveis automáticas durante a execução de um programa. No entanto, ela pode ser omitida numa primeira leitura.

A alocação dos espaços de memória dos parâmetros e das variáveis locais seguem um *modelo de pilha*. Podemos fazer uma analogia com uma pilha de pratos. Quando queremos adicionar um prato na pilha, este prato só pode ser colocado no topo da pilha. Se quisermos tirar pratos da pilha, só podemos tirar os pratos que estão no topo da pilha. O sistema gerencia a memória das

variáveis automáticas da mesma maneira. Uma área da memória do computador é reservada para armazenar a *pilha de execução* do programa. Quando a declaração de uma variável é encontrada, o espaço de memória no topo da pilha de execução é associado à variável. Quando o tempo de vida da variável se extingue, o espaço correspondente do topo da pilha é liberado para ser usado por outra variável.

Para ilustrar o modelo de pilha, vamos usar um esquema representativo da memória do computador usado para armazenar a pilha de execução. A Figura 1 ilustra a alocação de variáveis automáticas na pilha durante a execução do programa que faz o cálculo do volume de um cilindro, reproduzido abaixo:

```
#include <stdio.h>

#define PI  3.14159

float volume_cilindro (float r, float h)
{
   float v;
   v = PI * r * r * h;
   return v;
}

int main (void)
{
   float raio, altura, volume;
   printf("Entre com o valor do raio: ");
   scanf("%f", &raio);
   printf("Entre com o valor da altura: ");
   scanf("%f", &altura);

   volume = volume_cilindro(raio,altura);

   printf("Volume do cilindro = %f", volume);
   return 0;
}
```

Quando a execução do programa se inicia, a função *main* começa sua execução. No nosso exemplo, no início da função, três variáveis são declaradas (Figura 1(a)). As três variáveis são então alocadas no topo da pilha (inicialmente vazia). Como nenhum valor é atribuído às variáveis, os espaços de memória correspondente armazenam valores indefinidos ("lixos"). Em seguida, o programa requisita que o usuário defina os valores do raio e da altura do cilindro. Estes valores capturados são atribuídos às variáveis correspondentes (Figura 1(b)). Após a captura dos valores, a função *main* invoca a função auxiliar para realizar o cálculo do volume. A chamada da função representa primeiramente uma transferência do fluxo de execução para a função. Os parâmetros da função são alocadas na pilha e seus valores são inicializados com os valores passados na chamada da função (Figura 1(c)). Note que neste momento, como o controle da execução foi transferido para a função auxiliar, não se tem acesso às variáveis declaradas dentro da função *main*, apesar delas ainda estarem alocadas na base da pilha (a função *main* ainda não terminou, apenas teve a sua execução suspensa). Em seguida, dentro da função auxiliar, uma variável local adicional é declarada, sendo alocada no topo da pilha de execução, inicialmente com valor indefinido (Figura 1(d)). A função auxiliar então computa o valor do volume e atribui à variável local (Figura 1(e)). Este valor é então retornado pela função. Neste momento, a função auxiliar termina sua execução e o controle volta para a função *main*. Os parâmetros e variáveis locais da função auxiliar são desempilhadas e deixam de existir. Finalmente, já no retorno do controle para a função *main*, o valor retornado pela função é atribuído à variável

`volume` (Figura 1(f)).

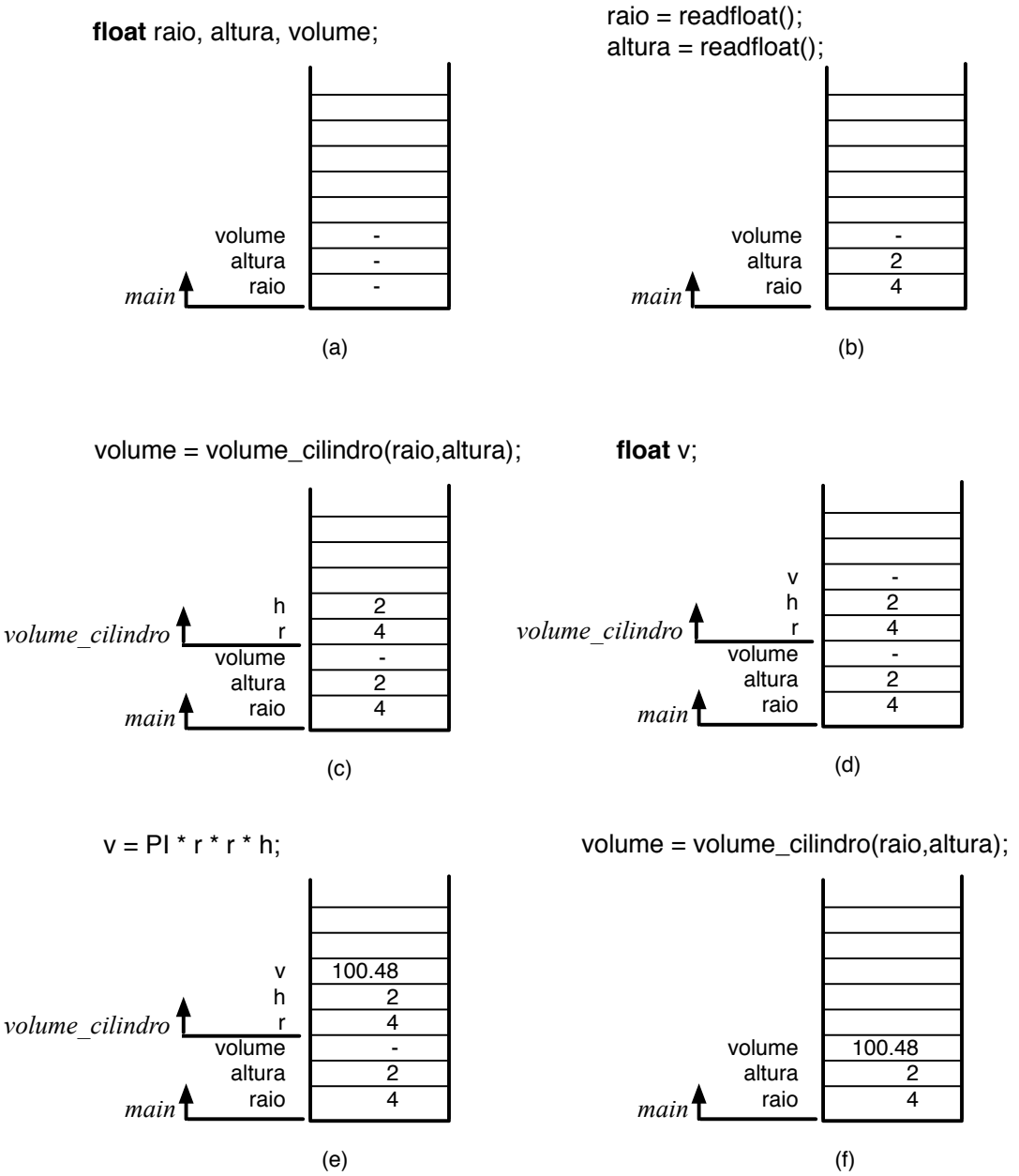


Figura 1: Pilha de variáveis durante execução de um programa.

Este exemplo simples serve para ilustrar o funcionamento do modelo de pilha. Logicamente, programas reais são mais complexos. Em geral, um programa tem várias funções auxiliares. Funções auxiliares podem evidentemente chamar outras funções auxiliares, sempre seguindo o modelo de pilha. Ao término da execução de cada função, as variáveis locais correspondentes são desempilhadas e o controle volta para a função que chamou.

## 5 Trabalhando com várias funções

Para exemplificar a organização de programas em diferentes funções, vamos considerar o problema de computar propriedades geométricas de modelos obtidos por composição. Para ilustrar, vamos considerar o projeto de uma nova peça mecânica. Para se projetar uma peça é necessário

a existência de uma metodologia de modelagem. Um processo de modelagem muito usado é a criação de modelos por composição. Um novo objeto pode ser modelado através de vários objetos simples, combinados através de operações booleanas (união, interseção e diferença). A Figura 2 ilustra um modelo de uma peça com furos em forma de L, obtida por uma união de duas caixas (paralelogramas) e duas diferenças de cilindros: a primeira caixa tem dimensão $b \times h \times e$ e a segunda tem dimensão $b \times (b - e) \times e$; os dois cilindros têm altura $e$ e raio $\frac{d}{2}$.

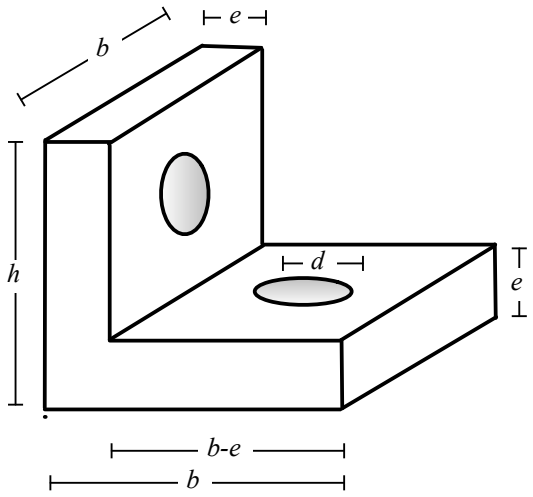


Figura 2: Exemplo de objeto modelado por composição.

Vamos considerar o problema de calcular as propriedades geométricas desta peça. Em particular, queremos calcular o volume e a area da superfície externa. Em resumo, queremos construir um programa que capture as dimensões da peça fornecidas via teclado (altura ($h$), base ($b$), espessura ($e$) e diâmetro do furo ($d$)) e exiba o volume e a área da superfície externa da peça na tela. Para tanto, podemos pensar em organizar nosso programa em pequenas funções. Inicialmente, vamos pensar na codificação de funções para o cálculo de propriedades físicas de objetos simples. Para a peça em questão, identificamos a necessidade de computar as seguintes propriedades:

- volume de caixa (paralelogramo)
- volume de cilindro
- área externa de caixa
- área externa de cilindro
- área de retângulo
- área de círculo

Uma função para computar o volume de uma caixa pode ser expressa por:

```
float volume_caixa (float a, float b, float c)
{
   float v;
   v = a * b * c;
   return v;
}
```

onde a, b e c representam os lados da caixa. A mesma função pode ser escrita de forma mais concisa, por:

```
float volume_caixa (float a, float b, float c)
{
   return a * b * c;
}
```

As demais funções podem ser codificadas de forma similar:

```
#define PI 3.14159


float volume_cilindro (float r, float h)
{
   return PI * r * r * h;
}

float area_caixa (float a, float b, float c)
{
   return 2 * (a*b + b*c + c*a);
}

float area_cilindro (float r, float h)
{
   return 2 * PI * r * h;
}

float area_retangulo (float a, float b)
{
   return a * b;
}

float area_circulo (float r)
{
   return PI * r * r;
}
```

Com estas funções, podemos construir nossa função principal. O volume da peça ilustrada na Figura 2 pode ser obtido somando os volumes das duas caixas e diminuindo o volume dos dois cilindros. A área da superfície externa pode ser feito somando as áreas das duas caixas e as áreas dos dois cilindros, e diminuindo duas áreas retangulares (área de contato entre as duas caixas) e quatro áreas de círculos ("tampos" dos cilindros). A codificação desta função é ilustrada a seguir.

```
int main (void)
{
   float h, b, e, d;     /* dimensões da peça */
   float v, s;           /* volume e área da superfície */

   /* captura valores fornecidos via teclado */
   printf("Entre com a altura: ");
   scanf("%f", &h);
   printf("Entre com a base: ");
   scanf("%f", &b);
```

```
    printf("Entre com a espessura: ");
    scanf("%f", &e);
    printf("Entre com o diametro dos furos: ");
    scanf("%f",&d);

    /* cálculo do volume da peça */
    v = volume_caixa(b,h,e) + volume_caixa(b-e,b,e) - 2*volume_cilindro(d/2,e);

    /* cálculo da área da superfície externa */
    s = area_caixa(b,h,e) + area_caixa(b,b-e,e) + 2*area_cilindro(d/2,e)
        - 2*area_retangulo(b,e) - 4*area_circulo(d/2);

    /* exibe na tela os valores computados */
    printf("Volume = %f e Area = %f \n", v,s);

    return 0;
}
```

## Exercícios

1. No programa para cálculo de volume e área da superfície da peça em forma de L, crie duas funções auxiliares adicionais para calcular o volume e a área, respectivamente, de uma caixa com um furo cilíndrico. As funções devem receber como parâmetros as dimensões da caixa com furo: altura, base, espessura e diâmetro do furo.

2. Usando as funções do item anterior, re-escreva a função *main* para que ela utilize estas funções.

3. Escreva um programa, estruturado em diversas funções, para calcular o volume de uma peça formada por uma esfera com um furo cilíndrico, dados os valores de $d$ e $D$ conforme ilustrado na Figura 3. Sabe-se que o volume de uma calota esférica de altura $h$ é dada por $\frac{1}{3}\pi h^2(3R - h)$, onde $R$ representa o raio da esfera.

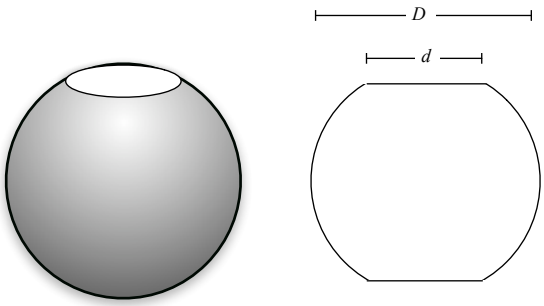


Figura 3: Modelo de uma esfera com furo cilíndrico: vistas 3D e 2D.