Programação em C

Francisco de Assis Boldt

22 de maio de 2019

Sumário

Capítulo 1

Programas em C

Um algoritmo é uma sequência finita de ações que corresponde a um padrão de comportamento [Dijkstra, 1971]. Quando um algoritmo é escrito em uma linguagem de programação, ele pode ser executado como um programa. O menor programa completo em C é apresentado no programa 1. Na verdade, programas menores, que usam menos código, podem ser feitos retirando-se a palavra `int` e a linha `return 0;`. Porém, tal programa estaria fora do padrão aceito por qualquer arquitetura, sistema operacional e compilador.

Programa 1: faznada.c

```
1 int main(){
2     return 0;
3 }
```

C é uma linguagem tipicamente compilada. Isto é, usando-a da forma que foi idealizada, é necessário que se "traduza" o programa para a linguagem de máquina (binário) para que ele seja executado. O console 1 mostra como compilar e executar o programa 1, usando um dos compiladores mais populares, o gcc [GNU Project, 2019]. A linha 3 do console 1 mostra o prompt esperando o próximo comando.

Console 1: Compilando e executando o programa 1.

```
1 $ gcc faznada.c -o faznada.bin
2 $ ./faznada.bin
3 $
```

Apesar do programa gerado pelo código do algoritmo 1 não apresentar nada na tela quando executado, para o sistema operacional (SO) este programa faz alguma coisa. O SO precisa reservar um espaço de memória e tempo de uso do processador para este programa. Além disso, o SO também espera o fim da execução de qualquer programa e exige um código de erro, que é um número inteiro. Quando o programa executa sem erros o código retornado é 0 (zero). Este é o motivo pelo qual o algoritmo 1 inicia com `int`. O `return 0;` na linha 2 diz para o SO que o programa foi executado com sucesso. Em geral, os comandos em C terminam com um ponto e vírgula.

O início e o fim das funções em C são sinalizados por abertura ({) e fechamento (}) de chaves, respectivamente. A abertura e fechamento de parênteses após o nome da função também é obrigatória. Dentro dos parênteses são declarados os parâmetros da função. A palavra `main` indica que esta é a função principal do programa. No caso do algoritmo 1 é a única função do programa. Mesmo quando um arquivo fonte escrito em C contém várias funções, a função `main` sempre será a primeira a ser chamada pelo SO quando o programa for executável.

Os programas são chamados de executáveis quando podem ser ativados por uma ação direta do usuário. A maioria dos programas que rodam em um computador moderno são ativados por outros programas, e não por uma ação direta do usuário. Esses não são considerados programas executáveis e, consequentemente, não precisam de um função `main`. Normalmente, tais programas estão em bibliotecas usadas por programadores.

1.1 printf

A linguagem C possui várias bibliotecas para ajudar os programadores. Uma delas é a biblioteca de entrada e saída padrão (`stdio.h` - STanDard Input and Output). Esta biblioteca oferece a função `printf`, que exibe uma cadeia de caracteres no terminal. Antes de usar uma biblioteca precisamos incluí-la no programa utilizando a diretiva de compilação `#include`, colocando-se o nome da biblioteca entre os sinais de < e >, como pode ser visto no programa 2.

Programa 2: hello.c

```
1 #include <stdio.h>
2 int main() {
3     printf("Hello World!\n");
4     return 0;
5 }
```

O programa gerado por este código imprime a frase "Hello World!" na tela do computador. A abertura e o fechamento das aspas na linha 3 indica que o conteúdo entre elas é uma cadeia de caracteres. O `\n` indica um quebra de linha no final da frase. O console 2 mostra como compilar e executar o programa 2, assim como o seu resultado. A linha 3 mostra o resultado do programa e a linha 4 mostra o prompt esperando o próximo comando.

Console 2: Compilando e executando o programa 2.

```
1 $ gcc hello.c -o hello.bin
2 $ ./hello.bin
3 Hello World!
4 $
```

Programas de computadores executam essencialmente operações matemáticas. Operações como soma podem ser executadas, como mostrado no programa 3. O programa gerado com este código imprime "5 + 7 = 12" na tela. O %d representa um número inteiro que deve vir depois da vírgula, que neste caso é 12. A linha 2 não é compilada nem executada. Na linguagem C, tudo que fica entre "/*" e "*/" é um comentário. Comentários servem para orientar os programadores sobre o que o programa faz sem precisar analisar o código para descobrir. No momento, pode parecer perda de tempo escrever algo aparentemente desnecessário. Entretanto, para programas maiores, com dezenas, centenas ou até milhares de linhas de código, comentários podem poupar bastante tempo, já que esses códigos grandes geralmente não são triviais de se entender apenas lendo-os.

Programa 3: soma5e7.c

```
1 #include <stdio.h>
2 /* Programa que imprime a soma de 5 e 7 */
3 int main() {
4     printf("5 + 7 = %d\n", 5+7);
5     return 0;
6 }
```

A operação de soma é computada fora das aspas, depois da vírgula. Se a linha quatro for alterada para `printf("5 + 7 = %d\n", 5*7);`, o resultado impresso no console será `5 + 7 = 35`. Isso por que o sinal de * é o operador de multiplicação em C e na maioria das linguagens de programação. Portanto, pouco importa, para efeito de cálculo, o que está dentro das aspas. O conteúdo das aspas apenas formata o que será apresentado no terminal. O cálculo é feito depois da vírgula e o resultado é apresentado no lugar do `%d`.

1.2 Variáveis

Podemos notar que se precisarmos alterar o programa 3, por exemplo trocando de 7 para 8, teremos que trocar em dois lugares. Isso não parece ser algo prático, principalmente se tivermos fórmulas mais complexas do que uma simples soma. O programa 3 pode ficar um pouco mais reutilizável o implementarmos como mostra o programa 4. Com este algoritmo, caso queiramos mudar de 7 para 8, basta alterarmos a linha 6. Veja que neste caso temos `"%d + %d = %d\n"` ao invés de `"5 + 7 = %d\n"`. Agora, são necessários três números, um para cada `%d`. Os números são associados aos `%d`'s na ordem em que são apresentados.

Programa 4: somaxy.c

```
1 #include <stdio.h>
2 /* Programa que imprime a soma de duas variaveis */
3 int main() {
4     int x, y;
5     x = 5;
6     y = 7;
7     printf("%d + %d = %d\n", x, y, x+y);
8     return 0;
9 }
```

Para usarmos variáveis em C, precisamos declará-las antes. É assim que pedimos ao SO para reservar um espaço de memória para nossos programas. As variáveis em C possuem tipos com tamanhos diferentes. Então o tipo da variável influencia na quantidade de memória reservada para o programa. A declaração de um número inteiro é feita usando-se a palavra `int` seguida do nome da variável, que deve começar com uma letra. A linguagem C faz distinção entre letras maiúsculas e minúsculas. Uma boa prática de programação é usar nomes de variáveis que possuem relação lógica com a utilidade da variável. Caracteres que não são alfanuméricos devem ser evitados, assim como caracteres acentuados. Uma exceção é o caractere '_', que é muito utilizado em nomes de variáveis com mais de uma palavra, uma vez que espaços não são permitidos. O console 3 mostra que o resultado do programa 4 é idêntico ao resultado do programa 3. A diferença é que o programa 4 é mais fácil de ser alterado.

Console 3: Compilando e executando o programa 4.

```
1 $ gcc somaxy.c -o somaxy.bin
2 $ ./somaxy.bin
3 5 + 7 = 12
```

Tão difícil quanto implementar programas de computador é a manutenção deles. Portanto, os programas devem ser implementados de uma forma que sejam de fácil manutenção. São muito raros os programas que não precisam ser atualizados. Veja como programas comerciais sempre estão lançando novas versões (versão x.y.z). Por isso, deixar comentários no código é tão importante para poupar tempo e evitar novos erros.

O programa 5 mostra o algoritmo `somaxy` alterado. Agora, atribuindo valores às variáveis na declaração.

Programa 5: somaxy.c alterado

```
1 #include <stdio.h>
2 /* Programa que imprime a soma de duas variaveis */
3 int main() {
4     int x = 5, y = 8;
5     printf("%d + %d = %d\n", x, y, x+y);
6     return 0;
7 }
```

1.3 Tipos de Variáveis

A linguagem C possui dois tipos primitivos de variáveis divididos em duas categorias: inteiros e de ponto flutuante (reais). Apesar de cada categoria ter vários tipos com características diferentes, por enquanto usaremos basicamente dois tipos: `int` para números inteiros e `float` para números reais. Uma lista completa com os tipos primitivos em C pode ser encontrada em `https://en.wikipedia.org/wiki/C_data_types` [2019]. O processador do computador usa partes diferentes de sua unidade lógica e aritmética para fazer cálculos com números inteiros e de ponto flutuante. A função printf também diferencia o formato de números inteiros e de ponto flutuante. Veja o que acontece no programa 6 quando trocamos o tipo da variável na declaração, sem alterar o formato na impressão do resultado.

Programa 6: Imprimindo float com %d (ERRADO)!

```
1 #include <stdio.h>
2 /* Programa que imprime a soma de duas variaveis */
3 int main() {
4     float x = 5;
5     float y = 7;
6     printf("%d + %d = %d\n", x, y, x+y);
7     return 0;
8 }
```

O console 4 mostra o que acontece ao compilar o programa 6. A primeira coisa estranha que acontece é durante a compilação do programa, que é chamada na linha 1 do console. Três avisos ocorrem, um para cada %d. São avisos de que algo pode não sair como o esperado. E realmente não sai. Note que na linha 9 do console aparece -224424600 no lugar de 5, -224424584 no lugar de 7 e o resultado aparece como 0 e não 12! O %d é para imprimir números do tipo `int`. Por isso, quando o programa compilado é chamada na linha 8, o resultado da linha 9 é totalmente diferente do que poderíamos esperar. É importante ressaltar que erros e avisos de compilação são mais fáceis de se corrigir do que erros de lógica, como veremos nas seções seguintes.

Console 4: Compilando e executando o programa 6.

```
1  $ gcc somaxy.c -o somaxy.bin
2  somaxy.c: In function 'main':
3  somaxy.c:7:12: warning: format '%d' expects argument of type 'int', but
       argument   2 has type 'double' [-Wformat=]
4        printf("%d + %d = %d\n", x, y, x+y);
5               ^
6  somaxy.c:7:12: warning: format '%d' expects argument of type 'int', but
       argument   3 has type 'double' [-Wformat=]
7  somaxy.c:7:12: warning: format '%d' expects argument of type 'int', but
       argument   4 has type 'double' [-Wformat=]
8  $ ./somaxy.bin
9  -224424600 + -224424584 = 0
10 $
```

Vamos entender o que os avisos (*warning*) significam. A linha 2 do console mostra que existe algo inesperado na função `main` no código fonte `somaxy.c`. Isso ainda não parece importante porque nós só tivemos a função `main` dentro de apenas um arquivo de código fonte até o momento. Porém, quando temos várias funções em vários arquivos, saber em que função e arquivo está o erro ou aviso ajuda bastante. A linha 3 do console mostra um aviso (*warning*) de que algo estranho está no arquivo somaxy.c na linha 7 e coluna 12. A linha 4 do console mostra o local exato do ocorrido com um sinal `^`. Traduzindo, o formato '%d' espera argumento do tipo 'int', mas o argumento 2 é do tipo 'double'. A explicação do aviso é a mesma para as linhas 6 e 7, exceto o número do argumento que é 3 e 4, respectivamente. Note que o compilador mostra que o erro está na coluna 12 para os três %d. É uma indicação aproximada de onde o erro ocorreu.

A formatação correta para números do tipo `float` é %f, como mostra o programa 7.

Programa 7: Imprimindo float com %f.

```
1 #include <stdio.h>
2 /* Programa que imprime a soma de duas variaveis */
3 int main() {
4     float x = 5, y = 7;
5     printf("%f + %f = %f\n", x, y, x+y);
6     return 0;
7 }
```

O console 5 mostra o resultado do programa 7 com a formatação adequada para números do tipo `float`.

Console 5: Compilando e executando o programa 7.

```
1 $ gcc somaxy.c -o somaxy.bin
2 $ ./somaxy.bin
3 5.000000 + 7.000000 = 12.000000
4 $
```

Veja que agora os números aparecem com 6 casas decimais depois do ponto (padrão inglês dos Estados Unidos). A quantidade de casas decimais que o printf imprime não tem relação com a quantidade de casas decimais que é armazenada na memória do computador. Cada tipo da categoria de ponto flutuante possui um limite de menor número absoluto que pode ser armazenado, que não pode ser alterado a gosto do programador. Estes limites podem ser encontrados na biblioteca float.h [2019]. Por exemplo, o tipo `double` ocupa o dobro de espaço na memória do computador do que o tipo `float`. Ambos servem para armazenar números reais, mas o tipo `double` é mais preciso, sendo capaz de armazenar números muito grandes ou muito próximos de zero. Por outro lado, a formatação de impressão dos números pode ser facilmente alterada. Por exemplo, se quisermos apenas duas casas decimais depois do ponto, basta substituirmos os %f por %.2f. Um referência completa dos formatos aceitos pela função printf pode ser encontrada em `http://www.cplusplus.com/reference/cstdio/printf` [2019].

1.4 Operadores

Nesta seção veremos apenas dois tipos de operadores. O operador de atribuição e os operadores aritméticos. Demais operadores da linguagem C serão vistos nos capítulos seguintes.

1.4.1 Operador de atribuição

Como já foi visto, mas não explicado nas seções anteriores, para atribuirmos um valor a uma variável é necessário o operador de atribuição (=). Note que apenas uma variável deve estar do lado esquerdo do sinal de igualdade (*variavel = valor*). O lado direito pode ter uma expressão que resulte em um valor do mesmo tipo da variável (explicado na seção seguinte). Mas, uma atribuição é uma expressão que resulta em um valor. Por isso, é possível fazer atribuições múltiplas.

Exemplo:

```
int main() {
    int x, y;
    x = y = 7;
    return 0;
}
```

A variável y recebe o valor 7. Essa atribuição resulta no mesmo valor, que é atribuído à variável x. O resultado é que tanto x quanto y acabam armazenando o valor 7.

1.4.2 Operadores aritméticos

Esta seção não apresenta todos os operadores aritméticos da linguagem C. Apresenta apenas os necessários para desenvolver os problemas propostos nas seções seguintes.

O operador `+` já foi apresentado nas seções anteriores. Ele resulta na soma de dois valores numéricos, sejam eles variáveis ou não. Exemplo: `variavel1 = variavel2 + 10`. No exemplo a `variavel2` tem seu valor somado ao valor 10. O resultado dessa soma é atribuído à `variavel1`. Uma observação importante é que se um dos operandos forem do mesmo tipo, o resultado é do mesmo tipo dos operandos. Se um valores do tipo `int` e outro do tipo `float`, o resultado será do tipo `float`. O resultado será sempre do tipo que ocupa mais espaço na memória do computador. Ou seja, se um operando for `float` e o outro for `double`, o resultado será do tipo `double`.

O operador `*` é o operador de multiplicação funciona de forma análoga ao operador `+`, exige dois operandos e retorna o produto deles.

O operador `-` tem duas funções. Como operador binário ele tem dois operandos e funciona de forma análoga ao operador `+`, subtraindo o segundo operando do primeiro. O tipo do resultado segue a mesma lógica do operador `+`. Como operador unário ele só tem um operando, que é multiplicado por `-1`.

O operador `/` é o operador de divisão e possui duas funções. Quando um dos operando é de ponto flutuante (`float` ou `double`), ele funciona como uma divisão de ponto flutuante e resulta em um valor do maior tipo. Quando os dois operandos são números inteiros (`int`), este operador resulta na divisão inteira dos dois números. Então, a divisão `10/4` é igual a `2` e não `2.5`.

Um operador muito importante em programação é o operador `%`, que retorna o resto da divisão entre dois números inteiros. Este operador exige dois operandos de tipo inteiro e retorna o resto da divisão do primeiro operando pelo segundo. Então, o resultado da operação `7%3` é `1`.

Os operadores em linguagem C possuem precedências similares à matemática. Operadores de multiplicação e divisão (`*, /, %`) têm precedência aos operadores de soma e subtração (`+, -`). Porém o operador `-` unário tem precedência aos operadores de multiplicação e divisão. Se o programador quiser mudar a precedência dos operadores ou apenas deixar a precedência dos operadores mais clara, parênteses podem ser usados.

Demais operadores aritméticos da linguagem C serão apresentados nos capítulos seguintes.

1.5 Exercícios

Esta seção apresenta exercícios com solução comentada e exercícios propostos. Tente resolver os exercícios sem olhar a correção comentada. Caso não consiga resolver sozinho, veja a solução apresentada e depois tente resolver novamente sem olhar a solução.

1.5.1 Resolvidos

As soluções apresentadas a seguir são apenas sugestões. Outras soluções, também corretas, podem ser desenvolvidas. Se sua resposta não estiver igual à solução apresentada, não significa que sua resposta está errada. Por enquanto, o foco será no resultado impresso pelo programa.

1. Implemente um programa que imprima o quadrado de um número.

2. Implemente um programa para converter um valor em polegadas para seu correspondente em milímetros. A fórmula de conversão é 25,4 mm $\approx$ 1 polegada.

Exercício 1: A primeira decisão ao implementar este exercício é sobre o tipo da variável. No caso desse exercício, podemos fazer a forma mais geral, usando o tipo `float` (linha 3). A variável x é declarada e o valor 5 é atribuído a ela. Na linha 4, a variável quadrado é declarada e a expressão `x*x` é atribuída a ela. A expressão `x*x` em C é correspondente à notação matemática $x \times x$, que equivale a x^2. Na linha 5, a função printf mostra o resultado formatado.

```
1 #include <stdio.h>
2 int main() {
3     float x = 5;
4     float quadrado = x*x;
5     printf("%f^2 = %f\n", x, quadrado);
6     return 0;
7 }
```

Após compilar e executar o programa podemos verificar o resultado.

```
1 $ gcc quadrado.c -o quadrado.bin
2 $ ./quadrado.bin
3 5.000000^2 = 25.000000
4 $
```

Exercício 2: Mais uma vez o tipo `float` foi utilizado. Na linha 3, a variável pol recebe o valor 5 representando a quantidade de polegadas. Na linha 4, a variável mm recebe o valor convertido. A linha 5 imprime o resultado.

```
1 #include <stdio.h>
2 int main(){
3     float pol = 5;
4     float mm = pol*25.4;
5     printf("%f\" = %fmm\n", pol, mm);
6     return 0;
7 }
```

A compilação e execução deste exercício é similar à do exercício anterior.

```
1 $ gcc polegadas.c -o polegadas.bin
2 $ ./polegadas.bin
3 5.000000" = 127.000000mm
4 $$
```

1.5.2 Propostos

1. Implemente um programa que converta de pés para metros (1 pé $\approx$ 0.3048 metro).
2. Implemente um programa que converta de km para milhas (1 milha $\approx$ 1.60934 km).
3. Implemente um programa que calcule a área de um retângulo.
4. Implemente um programa que calcule o perímetro de um retângulo.
5. Implemente um programa que calcule a área de um círculo.
6. Implemente um programa que calcule o perímetro de um círculo.
7. Implemente um programa que calcule a função $x^3 + 3x^2 + 2x - 7$.

1.6 Constantes

Suponha que você precise implementar um programa para converter uma temperatura em graus Celsius para Fahrenheit. A fórmula de conversão é $f = \frac{9}{5} \times c + 32$, onde f representa a temperatura em Fahrenheit e c a temperatura em Celsius. Um programa 8 que parece correto, mas não é.

Programa 8: Converte Celsius para Fahrenheit ERRADO!

```
1 #include <stdio.h>
2 /* Converte temperatura em Celsius para Fahrenheit. */
3 int main(){
4   float f, c = 25;
5   f = 9/5*c+32;
6   printf("%f Celsius = %f Fahrenheit\n", c, f);
7   return 0;
8 }
```

Veja a execução do programa 8 no console 6. 25 graus Celsius correspondem a 77 graus Fahrenheit, mas o programa converteu para 57.

Console 6: Compilando e executando o programa 8. Resultado ERRADO!

```
1 $ gcc cel2fah.c -o cel2fah.bin
2 $ ./cel2fah.bin
3 25.000000 Celsius = 57.000000 Fahrenheit
4 $
```

O erro está na linha 5 do programa 8. Mais especificamente na divisão 9/5. Experimente mudar a expressão 9/5 para 1.8. Matematicamente são expressões equivalentes, mas o programa só funciona corretamente com 1.8. Um dos motivos para este erro ocorrer é o uso de constantes.

O programa 8 tem quatro números digitados: 25, 9, 5 e 32. Para que o processador use estes números em algum cálculo, eles devem estar na memória do computador, mas não podem ser modificados. Por isso, são chamados de constantes. Se estão na memória, precisam ter um tipo especificado. Mesmo linguagens de programação que não exigem declaração de variáveis possuem tipos, mas o programador não lida explicitamente com eles. Não é o caso da linguagem C, na qual o programador é responsável pelos tipos das variáveis e constantes. Em C, constantes numéricas sem ponto (como 25, 9, 5 e 32), são do tipo `int`. Se possuem ponto (como 1.8), são do tipo `double`. Se o programador quiser que uma constante com ponto seja do tipo `float`, ele deve colocar um `f` após o número (exemplo 1.8f).

O valor 25 é do tipo `int` mas a variável `c` é do tipo `float`. Então, quando o valor do tipo `int` 25 é atribuído a ela na linha 4 do programa, ele é convertido para `float`. Mas algo diferente acontece na linha 5. Existe expressão do lado direito da igualdade e não uma constante. Essa expressão precisa ser calculada em um valor do tipo `float` antes de ser atribuída à variável `f`. O cálculo das expressões em linguagem C possuem precedências que, em geral, são as mesmas da matemática. Por exemplo, multiplicação e divisão têm precedência em relação à soma e subtração. Em outras palavras, multiplicação e divisão vêm antes de soma e subtração. Entretanto, operadores de mesma precedência, como multiplicação (*) e divisão (/), são calculados da esquerda para direita.

Vamos fazer o passo a passo para calcular a expressão `9/5*c+32`. A soma tem precedência menor do que divisão e multiplicação, então será a última operação a ser feita. A expressão `9/5*c` possui divisão e multiplicação que têm mesma precedência, então serão executadas da direita para a esquerda, deixando a multiplicação para depois da divisão. A expressão `9/5` divide duas constantes inteiras. Mas a divisão de dois números inteiros em C resulta em um número inteiro. Se a divisão não for exata o número é truncado (arrendondado para baixo). No caso, `9/5=1`, e não 1.8. Assim, o valor 1 é multiplicada à variável `c`, que possui valor 25, resultando em 25. O valor 25 é somado ao valor 32, resultando em 57. Por isso, o programa funciona quando se troca `9/5` por 1.8. Outra forma de se fazer o programa corretamente seria mudar a expressão `f = 9/5*c+32` por `f = c*9/5+32`. Estas expressões são matematicamente equivalentes mas são diferentes em C.

1.7 Programação é Matemática

Suponha agora que você precise implementar um programa para converter uma temperatura em graus Fahrenheit para Celsius. Você já conhece a equação que faz a conversão contrária. A partir dela, você pode chegar na equação que faz a conversão desejada.

$$\begin{aligned} f &= \frac{9}{5}\times c + 32 \\ f - 32 &= \frac{9}{5}\times c \\ 5\times(f-32) &= 9\times c \\ \frac{5}{9}\times(f-32) &= c \\ c &= \frac{5}{9}\times(f-32) \end{aligned} \tag{1.1}$$

Com a equação 1.1 fica fácil implementar o programa. Tudo que se tem a fazer é "traduzir" a equação para linguagem C. Para programar é necessário ter consciência de que não adianta conhecer todos os comandos da linguagem de programação se a lógica para resolver os problemas não estiver desenvolvida. Lembre-se

que se você não consegue resolver um problema usando a matemática, não conseguirá resolvê-lo usando a linguagem C. Por isso, exercite sua lógica matemática e a lógica de programação evoluirá naturalmente. Veja como ficou o programa 9.

Programa 9: Converte Fahrenheit para Celsius.

```
1 #include <stdio.h>
2 /* Converte temperatura em Fahrenheit para Celsius. */
3 int main(){
4   float c, f = 77;
5   c = (5.0f/9)*(f-32);
6   printf("%f Fahrenheit = %f Celsius\n", f, c);
7   return 0;
8 }
```

Importante notar na linha 5 do programa 9 a expressão `5.0f`. O programa funcionaria se a expressão fosse `5.0`, mas o seu resultado seria sempre zero se a expressão fosse `5`. Isso porque `5/9` dá zero, e multiplicado por qualquer outra coisa `(f-32)`, também daria zero. A expressão `5.0f` é preferida à `5.0` porque esta é uma constante do tipo `double` enquanto aquela é do tipo `float`. Considerando que a variável `c` é do tipo `float`, uma constante do tipo `float` pode evitar erros indesejáveis. Veja a execução do programa 9 no console 7.

Console 7: Compilando e executando o programa 9.

```
1 $ gcc fah2cel.c -o fah2cel.bin
2 $ ./fah2cel.bin
3 77.000000 Fahrenheit = 25.000002 Celsius
4 $
```

Note que o resultado da conversão de 77 Fahrenheit não resultou em exatos 25 graus Celsius. Devemos esperar algumas imprecisões quanto a divisões.

1.8 Exercícios

1. Implemente um programa que dada uma quantidade de segundos, calcule a quantidade de horas, minutos e segundos. Exemplo: 4000 s ==> 1 h, 6 min, 40 s.

2. Implemente um programa que dada um valor inteiro representando uma quantidade de dinheiro, calcule a quantidade mínima de notas para este valor. Por exemplo, a resposta para o valor 375 deve ser:

   ```
   3 Nota(s) de 100.00
   1 Nota(s) de 50.00
   1 Nota(s) de 20.00
   0 Nota(s) de 10.00
   1 Nota(s) de 5.00
   0 Nota(s) de 2.00
   0 Nota(s) de 1.00
   ```

3. Implemente um programa que calcule o preço final de um produto após um desconto dado em porcentagem. O programa deve imprimir o valor nominal do produto, seu desconto em dinheiro (não em porcentagem) e o preço final do produto.

4. Faça um algoritmo para calcular o valor a ser pago pelo período de estacionamento do automóvel, usando os seguintes dados: hora e minuto de entrada, hora e minuto de saída. Sabe-se que este estacionamento cobra R$ 4,00 por hora, mas calcula as frações de hora também. Por exemplo, se a pessoa ficar 1 hora e quinze minutos, pagará R$ 5,00 (R$ 4,00 pela hora e R$ 1,00 pelos quinze minutos).

Considerações do Capítulo

Este capítulo apresentou os conceitos e comandos mínimos para se implementar um programa em linguagem C. É um bom começo, mas programas úteis precisam de mais conceitos e comandos. Eles serão apresentados gradativamente no capítulos seguintes.

Destaques do capítulo:

- Um programa que é executado diretamente pelo usuário deve possuir uma função chamada `main`. Esta função deve iniciar com a palavra `int` e terminar com o comando `return 0;`.
- A função printf exibe uma mensagem na tela. Essa mensagem pode conter números que são resultado de operações matemáticas. Para usar a função printf é necessário incluir a biblioteca `stdio.h` no início do programa.
- Variáveis facilitam a programação. Elas possuem tipos e devem ser declaradas antes de ser usadas.
- Operadores matemáticos podem ter mais de uma função e seu resultado depende do contexto.
- Contantes também possuem tipo e seu uso incorreto pode acarretar em erros.
- Programação é matemática. Os problemas devem ser resolvidos antes de virar código.

Capítulo 2

Subprogramas: Funções e Procedimentos

Funções em C podem ser entendidas como pequenos programas e também podem ser chamadas de subprogramas. Normalmente as linguagens de programação fazem distinção entre funções e procedimentos, onde funções retornam algum valor enquanto procedimentos não.

2.1 Procedimentos

Um exemplo de procedimento é o subprograma que imprime "Hello world!" na tela, como mostra o programa 10. Em C, a diferença entre função e procedimento está no retorno da função. No exemplo do programa 10, a declaração do subprograma `hello` inicia com a palavra reservada `void`, indicando que este subprograma não retorna valor algum e, portanto, é um procedimento.

Programa 10: hello_sub.c

```
1  #include <stdio.h>
2  /* Procedimento que imprime a frase "Hello World!". */
3  void hello(){
4      printf("Hello World!\n");
5  }
6  int main() {
7      hello();
8      hello();
9      return 0;
10 }
```

Veja a compilação e execução do programa 10 no console 8.

Console 8: Compilação e execução do programa 10

```
1 $ gcc hello_sub.c -o hello_sub.bin
2 $ ./hello_sub.bin
3 Hello World!
4 Hello World!
5 $
```

Duas vantagens de se usar um procedimento é que uma vez feito, ele pode ser chamado quantas vezes necessário, e quando se altera o procedimento, todas suas chamadas também são alteradas. Copiar é colar código é uma péssima prática de programação e deve ser evitada a todo custo. O uso de subprogramas é a forma mais comum de evitar a repetição de código.

O programa 11 alterou somente a função `hello`, mas a alteração teve efeito na função `main`. Esta é uma das principais vantagens de usar subprogramas. A alteração do código fica concentrada em apenas um lugar.

Programa 11: hello_sub.c

```
1  #include <stdio.h>
2  /* Procedimento que imprime a frase "Ola Mundo!". */
3  void hello(){
4      printf("Ola Mundo!\n");
5  }
6  int main() {
7      hello();
8      hello();
9      return 0;
10 }
```

Veja a compilação e execução do programa 11 no console 9.

Console 9: Compilação e execução do programa 11

```
1  $ gcc hello_sub2.c -o hello_sub2.bin
2  $ ./hello_sub2.bin
3  Ola Mundo!
4  Ola Mundo!
5  $
```

2.2 Parâmetros

Programas são mais versáteis quando geram saídas diferentes dependendo da entrada. Por exemplo, o procedimento `hello` no programa 10 imprime sempre a mesma frase, o que o torna muito limitado. Muito mais interessante é o procedimento `imprime_soma` apresentado no programa 12. Nota-se que, depois depois de criado, o subprograma pode ser reutilizado quantas vezes for necessário. Ele gera resultados diferentes dependendo dos parâmetros passados.

Programa 12: somaxy_sub.c

```
1  #include <stdio.h>
2  void imprime_soma(int x, int y) {
3      printf("%d + %d = %d\n", x, y, x+y);
4  }
5  int main() {
6      imprime_soma(5, 7);
7      imprime_soma(6, 8);
8      imprime_soma(4, 9);
9      return 0;
10 }
```

Ao definir um subprograma, seja ele uma função ou um procedimento, devemos incluir a lista de parâmetros. No caso do procedimento `hello` no programa 10, a lista de parâmetros é vazia, pois não existe nada entre os parênteses colocados após o nome do procedimento. Por outro lado, o procedimento `imprime_soma` define uma lista com dois parâmetros inteiros, `int x` e `int y`. A definição de parâmetros de um subprograma se assemelha muito com a declaração de variáveis.

O console 10 mostra a compilação e execução do programa 12. Veja que além de ser uma boa prática de programação, utilizar subprogramas é mais fácil do que copiar, colar e alterar códigos.

Console 10: Compilação e execução do programa 12

```
1 $ gcc somaxy_sub.c -o somaxy_sub.bin
2 $ ./somaxy_sub.bin
3 5 + 7 = 12
4 6 + 8 = 14
5 4 + 9 = 13
6 $
```

Exercícios Resolvidos

1. Implemente o procedimento `imprime_multiplicacao`, que imprime o produto de dois números passados como parâmetro.
2. Implemente o procedimento `imprime_quadrado`, que imprime o produto de um número, passado como parâmetro, por ele mesmo.
3. Teste os procedimentos implementados em uma só execução dentro da função `main`.

Programa 13: parametros.c

```
1  #include <stdio.h>
2  void imprime_multiplicacao(int x, int y) {
3      printf("%d * %d = %d\n", x, y, x*y);
4  }
5  void imprime_quadrado(int x) {
6      imprime_multiplicacao(x, x);
7  }
8  int main() {
9      imprime_multiplicacao(5, 7);
10     imprime_quadrado(3);
11     return 0;
12 }
```

Como pode ser visto no programa 13, procedimentos podem usar outros procedimentos. O console 11 mostra a compilação e execução do programa 13.

Console 11: Compilação e execução do programa 13

```
1 $ gcc parametros.c -o parametros.bin
2 $ ./parametros.bin
3 5 * 7 = 35
4 3 * 3 = 9
5 $
```

2.3 Funções

As funções de linguagens de programação estão intimamente ligadas às funções matemáticas. Tomemos como exemplo o gráfico da função do segundo grau apresentada na figura 2.1. Qualquer linguagem de programação possui recursos para implementar a função da figura 2.1.

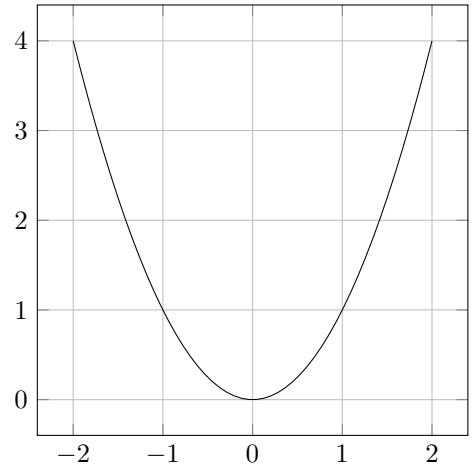


Figura 2.1: $f(x) = x^2$

A principal diferença entre procedimentos e funções é que funções retornam um valor. Este valor precisa ter um tipo. O tipo de retorno escolhido para a função `quadrado` foi `float`. O comando `return` e que define o valor que será retornado pela função. Este valor deve ser do tipo declarado antes no nome da função. O comando `return` é sempre o último comando executado por uma função. O programador pode até escrever algum código depois do comando, mas o código será ignorado. Em linguagem C esta implementação pode ser feita como mostra o programa 14.

Programa 14: quadrado.c

```
1  #include <stdio.h>
2  float quadrado(float x) {
3    return x*x;
4  }
5  int main() {
6    printf("f(%f) = %f\n", -2.0f, quadrado(-2));
7    printf("f(%f) = %f\n", -1.0f, quadrado(-1));
8    printf("f(%f) = %f\n", 0.0f, quadrado(0));
9    printf("f(%f) = %f\n", 1.0f, quadrado(1));
10   printf("f(%f) = %f\n", 2.0f, quadrado(2));
11   return 0;
12 }
```

Note que a função `main` só foi usada para testar a função `quadrado`. Entretanto, este teste ficou muito confuso. Por exemplo, a linha 6 tem dois parâmetros para a função `printf`, `-2.0f` e `quadrado(-2)`. Como o primeiro argumento da função `printf` é para imprimir um `%f`, seu tipo precisa ser `float`, senão um erro de impressão pode ocorrer. A constante `5` é do tipo `int` e a constante `5.0` é do tipo `double`. Nenhuma das duas satifaz o `%f`. Somente `5.0f`. Já o valor que a função quadrado recebe é convertido para `float` assim que a função é chamada. Por isso a contante `-2` pode ser usada.

2.3.1 Procedimentos de Teste

Podemos notar que as linhas 6 a 10 do programa 14 estão muito repetitivas. Isso é suscetível à equívocos. O ideal seria criar um procedimento para testar as funções. É claro que a função `quadrado` é muito simples, e é apresentada como um exemplo didático, mas funções mais complexas podem exigir testes. O programa 15 mostra como implementar uma função de teste.

Programa 15: testa_quadrado.c

```
1  #include <stdio.h>
2  float quadrado(float x) {
3    return x*x;
4  }
5  void testa_quadrado(float x) {
6    printf("f(%f) = %f\n", x, quadrado(x));
7  }
8  int main() {
9    testa_quadrado(-2);
10   testa_quadrado(-1);
11   testa_quadrado(0);
12   testa_quadrado(1);
13   testa_quadrado(2);
14   return 0;
15 }
```

O console 12 mostra a compilação e execução do programa 15.

Console 12: Compilação e execução do programa 15

```
1  $ gcc testa_quadrado.c -o testa_quadrado.bin
2  $ ./testa_quadrado.bin
3  f(-2.000000) = 4.000000
4  f(-1.000000) = 1.000000
5  f(0.000000) = 0.000000
6  f(1.000000) = 1.000000
7  f(2.000000) = 4.000000
8  $
```

Vale a pena ressaltar que nem todas funções precisam de ter um procedimento de teste. Como programador, você deve ter o bom senso de quando é melhor criar um procedimento para testar uma função ou quando é melhor testá-la diretamente na função `main`.

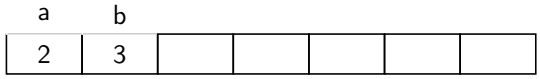
2.3.2 Escopo das Variáveis

Parâmetros são variáveis que têm seu valor atribuído quando os subprogramas são chamados. Estas variáveis só existem na memória do computador enquanto o subprograma está sendo executado, e são descartadas quando o subprograma termina. Por isso, os nomes das variáveis fora da função não precisam ser iguais ao dos parâmetros, como mostra o programa 16. Isso se chama escopo da variável. É a parte do código em que a variável é válida. Existe a diferença entre variáveis locais e globais, mas isso será tratado mais adiante. O uso de variáveis globais deve ser feito em casos muito específicos. Variáveis globais não devem ser usadas desnecessariamente, pois não é uma boa prática de programação.

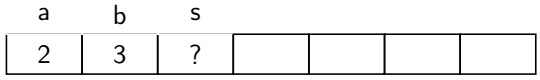
Programa 16: soma.c

```
1  #include <stdio.h>
2  float soma(float x1, float x2) {
3    return x1 + x2;
4  }
5  int main() {
6    float a = 2, b = 3;
7    float s;
8    s = soma(a, b);
9    printf("soma(%f, %f) = %f", a, b, s);
10   return 0;
11 }
```

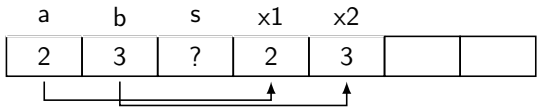
A figura 2.2 ajuda a ilustrar o que acontece na memória do computador quando o programa 16 é executado. O programa começa a ser executado na linha 5 com a declaração da função `main`. Na linha 6, as variáveis `a` e `b` são declaradas e os valores 2 e 3 são atribuídos a elas, respectivamente. A figura 2.2a ilustra a memória do computador com as posições de memória reservadas para as variáveis e seus valores já escritos. Na linha 7, a variável `s` é declarada, mas nenhum valor é atribuído a ela. Existe uma chance do valor dessa variável ser zero, mas isso não é nem um pouco garantido. É verdade que a variável `s` possui um valor. Mas esse valor é indeterminado. Por isso, uma interrogação foi colocada na posição de memória reservada para variável `s` na figura 2.2b. A atribuição de valor à variável `s` só acontece na linha 8. Porém, essa atribuição faz uma chamada à função `soma`. Então o programa continua sua execução na linha 2, onde as variáveis `x1` e `x2` são declaradas, reservadas na memória do computador e seus valores são atribuídos.



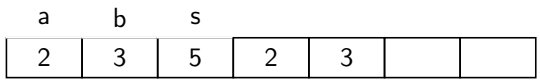
(a) Alocação e atribuição dos valores 2 e 3 às variáveis **a** e **b** (linha 6).



(b) Alocação sem atribuição de valor da variável **s** (linha 7).



(c) Chamada da função `soma` (linha 8 e 2).



(d) Retorno da função `soma` e atribuição de valor à variável `s` (linha 3 e 8).

Figura 2.2: Memória do computador durante a execução do programa 16.

Note na figura 2.2c que o valor da variável `s` ainda é indefinido até este momento. Note também que as variáveis `a` e `b` ocupam posições de memória distintas de `x1` e `x2`. Somente depois da execução do comando de retorno na linha 3 do programa é que a execução dele volta para linha 8 e o valor resultante da chamada

da função `soma` é atribuído à variável `s`, como ilustrado na figura 2.2d.

Fato importante de se entender é que, na figura 2.2d, as posições de memória que estavam reservadas para as variáveis `x1` e `x2` ainda estão com os valores 2 e 3. As variáveis não existem mais, porém seus valores ainda estão lá. Este é o motivo de uma variável ser alocada na memória sem valor definido. O valor armazenado na posição de memória reservada para uma variável pode ter sido usado anteriormente por outra variável e ainda conter seu valor antigo.

Na linha 9, o programa imprime os valores das variáveis `a`, `b` e `s`. Na linha 10 a função retorna o valor zero, indicando para o SO que o programa terminou corretamente, liberando todas as posições de memória reservadas para as variáveis.

2.3.3 Exercícios

Faça um programa que teste as funções abaixo:

1. Implemente uma função que multiplique dois números: `float produto(float x1, float x2);`.
2. Implemente uma função para converter uma polegada para milímetros. A formula de conversão é 25,4 mm = 1 polegada.
3. Implemente uma função para converter uma temperatura em graus Celsius para Fahrenheit. A fórmula de conversão é $f = \frac{9}{5} \times c + 32$, onde f representa a temperatura em Fahrenheit e c a temperatura em Celsius.
4. Implemente uma função para converter uma temperatura em graus Fahrenheit para Celsius.
5. Implemente uma função que receba dois números positivos representando os comprimentos dos lados de um retângulo e retorne a área desse retângulo.
6. Implemente uma função que receba um número positivo representando o lado de um quadrado e retorne a área dessa quadrado. Utilize a função anterior para implementar esta.
7. Implemente uma função que receba dois números positivos representando os lados de um retângulo e retorne seu perímetro.
8. Implemente um procedimento que receba dois números positivos representando os lados de um retângulo e imprima a área e o perímetro deste retângulo. Utilize as funções implementadas nos exercícios 5 e 7.

Solução da questão 1

O programa 17 mostra uma solução para o exercício 1. A solução apresentada é muito parecida com a do programa 16. A função `main` é necessária para testar o exercício.

Programa 17: produto.c

```
1 #include <stdio.h>
2 float produto(float x, float y) {
3   return x * y;
4 }
5 int main() {
6   float a = 2, b = 3;
7   printf("produto(%f, %f) = %f\n", a, b, produto(a,b));
8   return 0;
9 }
```

O console 13 mostra a compilação e execução do programa 17.

Console 13: Compilação e execução do programa 17.

```
1 $ gcc produto.c -o produto.bin
2 $ ./produto.bin
3 produto(2.000000, 3.000000) = 6.000000
4 $
```

2.3.4 Funções da biblioteca math.h

Como já mencionado na seção 1.1, a linguagem C possui várias bibliotecas para facilitar a programação. Uma biblioteca muito importante é a `math.h`[1]. Esta biblioteca oferece várias funções matemáticas comumente necessárias. Veja o exemplo apresentado no algoritmo 18.

Programa 18: cosseno.c

```
1  #include <stdio.h>
2  #include <math.h>
3  void testa_cos(double x) {
4     printf("cos(%f) = %f\n", x, cos(x));
5  }
6  int main() {
7     testa_cos(-2);
8     testa_cos(0);
9     testa_cos(1);
10    testa_cos(3.1415);
11    return 0;
12 }
```

Neste programa nós usamos a função `cos`[2] da biblioteca `math.h` para calcular o coseno (figura 2.3) de um número real. Para isso, precisamos de incluir esta biblioteca com a diretiva de compilação `#include`, assim como feito para a biblioteca `stdio.h`. Depois de incluída a biblioteca `math.h`, tanto a função `cos`, quanto as demais funções oferecidas por essa biblioteca, podem ser usadas como se tivessem sido implementadas no mesmo programa.

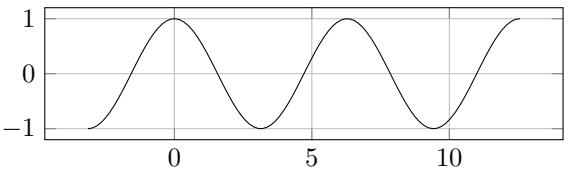


Figura 2.3: $f(x) = cos(x)$

O console 14 mostra a compilação e a execução do programa 18. É necessário o argumento "`-lm`" para compilar um programa que inclua a biblioteca `math.h`, como mostra a linha 1 do console.

Existem bibliotecas como gnuplot [2019] que fazem gráficos bonitos como o da figura 2.3. Mas a instalação de tais bibliotecas não serão apresentadas aqui. Testar funções geométricas como cosseno é bem complicado usando somente printf, como se vê no programa 18. Então, para testar mais facilmente essas funções, o programa 19 apresenta a função `plot2d`, que imprime o gráfico de uma função no terminal. A função `plot2d` não será explicada agora, somente como usá-la. Na linha 38 do programa 19 é chamada a função `plot2d`. Ela exige como primeiro argumento uma função com retorno do tipo `double`, com apenas um parâmetro, que deve ser do tipo `double`. O segundo parâmetro é o valor de início do eixo das abscissas e o terceiro parâmetro é seu valor final. O código fonte do programa 19 pode ser baixado em `https://github.com/fboldt/algoritmos/blob/master/plot2d.c`.

[1] http://www.cplusplus.com/reference/cmath/
[2] http://www.cplusplus.com/reference/cmath/cos/

Console 14: Compilação e execução do programa 18.

```
1 $ gcc cosseno.c -o cosseno.bin -lm
2 $ ./cosseno.bin
3 cos(-2.000000) = -0.416147
4 cos(0.000000) = 1.000000
5 cos(1.000000) = 0.540302
6 cos(3.141500) = -1.000000
7 $
```

O console 15 mostra a compilação e execução do programa 19. Veja que o gráfico da função cosseno não fica perfeito. Afinal, um gráfico contínuo exige um grau de detalhamento que o terminal não permite.

Programa 19: Função para gerar um gráfico 2D em um terminal.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  void plot2d(double (*function)(double), int domini, int domend) {
6    const double step = 0.25;
7    int *values, i, p, b, ymin=0, ymax=0;
8
9    domini = (int)round(domini/step);
10   domend = (int)round(domend/step);
11   values = (int*)malloc(sizeof(int)*(domend-domini));
12   for(i=domini; i<=domend; i++) {
13     p = i-domini;
14     values[p] = (int)(round(function(i*step)/step));
15     if (values[p] < ymin) ymin = values[p];
16     else if (values[p] > ymax) ymax = values[p];
17   }
18   for(i=ymax+1; i>=ymin-1; i--) {
19     printf("%8.2f ",i*step);
20     for(b=domini; b<=domend; b++) {
21       if(i == values[b-domini])
22         if(i==0) printf("O-");
23         else printf("O ");
24       else if(b == 0 && i == 0) printf("+-");
25       else if(b == 0) printf("| ");
26       else if(i == 0) printf("--");
27       else printf("  ");
28     }
29     printf("\n");
30   }
31   printf("    ");
32   for(i=domini; i<=domend; i+=4) printf("%8.2f", (i*step));
33   printf("\n");
34   free(values);
35 }
36
37 int main() {
38   plot2d(cos,-3, 5);
39   return 0;
40 }
```

Mas ainda assim, a visualização ficou bem melhor do que a do programa 18, onde quatro valores da função cosseno são impressos na tela.

Console 15: Compilação e execução do programa 19.

```
$ gcc plot2d.c -o plot2d.bin
$ ./plot2d.bin
    1.25                                |
    1.00                        O O O O O
    0.75                    O       |       O
    0.50                  O         |         O
    0.25                O           |           O                                 O
    0.00 ---------------O------------+-----------O-------------------------------O---
   -0.25             O              |             O                           O
   -0.50           O                |               O                       O
   -0.75       O O                  |                 O O               O O
   -1.00 O O                        |                     O O O O
   -1.25                            |
        -3.00    -2.00    -1.00     0.00     1.00     2.00     3.00     4.00     5.00
$
```

Função fabs

Outra função útil da biblioteca `math.h` é a `fabs`. Ela retorna o valor absoluto de um valor real. Para testá-la, altere a linha 38 do programa 19 para "`plot2d(fabs,-3, 3);`". Note que o valor final do eixo das abscissas foi reduzido para 3. Tente usar outros valores para início e fim do eixo das abscissas. O resultado será como o apresentado no console 16.

Console 16: Testando a função `fabs`.

```
$ gcc plot2d.c -o plot2d.bin -lm
$ ./plot2d.bin
    3.25                            |
    3.00 O                          |                              O
    2.75   O                        |                            O
    2.50     O                      |                          O
    2.25       O                    |                        O
    2.00         O                  |                      O
    1.75           O                |                    O
    1.50             O              |                  O
    1.25               O            |                O
    1.00                 O          |              O
    0.75                   O        |            O
    0.50                     O      |          O
    0.25                       O    |        O
    0.00 ----------------------------O-------------------------
   -0.25                            |
        -3.00    -2.00    -1.00     0.00     1.00     2.00     3.00
$
```

Função sqrt

Outra função importante é a `sqrt`, que retorna a raiz quadrada de um número real. Para testá-la lembre-se de que não existe raiz real para números negativos. Então, inicie o eixo das abscissas com zero, caso contrário, o gráfico não será impresso. Para testá-la, altere a linha 38 do programa 19 para "`plot2d(sqrt,0, 6);`".

2.3.5 Exercícios

Resolva os exercícios abaixo usando apenas operações aritméticas. Caso você já conheça comandos em C ou outra linguagem de programação que ainda não foram explicados em seções anteriores e podem facilitar a implementação dos exercícios a seguir, não use esses comandos. Estes exercícios são mais desafiadores quando resolvidos aritmeticamente. Para cada exercício faça um programa para testar a função implementada.

1. Implemente uma função que receba dois números reais positivos representando os catetos de um triângulo retângulo e retorne o comprimento da hipotenusa desse triângulo. Use a função `sqrt` da biblioteca `math.h`.

2. Implemente uma função que receba dois números positivos representando os catetos de um triângulo retângulo e retorne o perímetro desse triângulo. Utilize a função anterior para encontrar o terceiro lado do triângulo.

3. Implemente um procedimento que receba um número positivo representando o raio de um círculo e imprima a área e o perímetro desse círculo. Faça funções para calcular a área e o perímetro do círculo.

4. Faça uma função que receba um número inteiro x e um número inteiro n e retorne o algarismo da enésima casa decimal.
   Exemplos: `algarismo(1234, 1)==4` e `algarismo(1234, 2)==3`.

5. Faça uma função que troque a casa decimal de dois algarismos de um número.
   Exemplo: `troca_algarismos(6789, 2, 3)==6879`.

6. Faça uma função que retorne a primeira metade dos algarismos de um número.
   Exemplo: `metade1(1260)==12`.

7. Faça uma função que retorne a segunda metade dos algarismos de um número.
   Exemplo: `metade2(1260)==60`.

8. Analise a função da figura 2.4 e a implemente em linguagem C. (Este exercício é resolvido na próxima seção.)

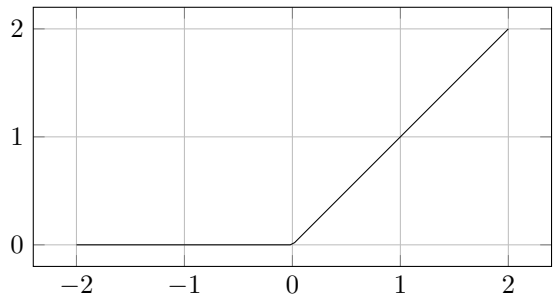


Figura 2.4: $f(x) = rampa(x)$

2.4 Estratégias para solução de problemas de programação

As estratégias apresentadas aqui são apenas sugestões. Cada programador tem sua forma de raciocinar para resolver os problemas.

2.4.1 Dividir para conquistar

A função da figura 2.4 retorna zero para todos os números menores que zero e retorna o próprio número quando este é maior que zero. Podemos ver claramente que o padrão muda quando o domínio da função cresce acima de zero. Esta é uma função por partes, então solucionar este problema dividindo-o em duas partes. A primeira parte para lida com números menores ou iguais a zero e a segunda, com números maiores que zero.

Se somarmos números negativos com seus valores absolutos teremos sempre zero, como pode ser observado na figura 2.5, onde a linha contínua representa a função identidade ($f_i(x) = x$), a linha tracejada representa

a função módulo ($f_m(x) = |x|$) e a linha pontilhada representa a soma dessas duas funções ($f_n(x) = f_i(x) + f_m(x)$). Isso é o que queremos para números menores que zero. Então, o retorno da função em C poderia conter o código `return x+abs(x);`. Isso resolve o problema parcialmente, pois temos zero quando o domínio é menor que zero, mas o valor para domínios positivos é sempre igual a $2 \times x$. O que nos leva para segunda parte da solução.

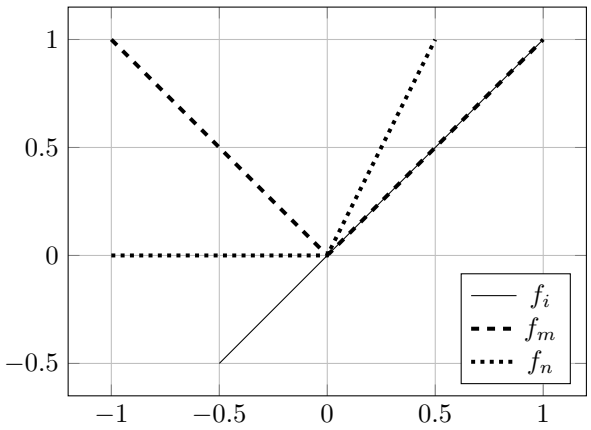


Figura 2.5: $f_i(x) = x$, $f_m(x) = |x|$, $f_n(x) = f_i(x) + f_m(x)$.

Como a função para números positivos é $f_p(x) = 2x$, se aplicarmos a função inversa ($f_p^{-1}(x) = \frac{x}{2}$) na parte positiva do domínio teremos o valor desejado. Mas, zero dividido por dois é sempre zero. Então esta mudança não afeta a primeira parte da solução. Assim, a função pode ser escrita como $f(x) = \frac{x+|x|}{2}$. A solução final é apresentada no algoritmo 20. Use o procedimento `plot2d` apresentado no programa 19 para verificar a solução apresentada.

Programa 20: rampa.c

```
1  double rampa(double x) {
2    return (x+fabs(x))/2;
3  }
```

O console 17 mostra a função rampa do programa 20 exibida pelo procedimento `plot2d`.

Console 17: Função rampa (programa 20) exibida pelo procedimento `plot2d`.

```
1       2.00                    |                    O
2       1.75                    |                  O
3       1.50                    |                O
4       1.25                    |              O
5       1.00                    |            O
6       0.75                    |          O
7       0.50                    |        O
8       0.25                    |      O
9       0.00 O-O-O-O-O-O-O-O-O-O-----------------
10     -0.25                    |
11          -2.00     -1.00     0.00      1.00      2.00
```

Note que o console 17 não apresenta um gráfico com a mesma qualidade da figura 2.4, mas permite verificar que a implementação da função esta correta.

2.4.2 Relacionar problemas novos a problemas já resolvidos

Suponha que você precise implementar a função na figura 2.6 em C. Você pode seguir o processo de desenvolvimento mostrado para implementar a função `rampa`.

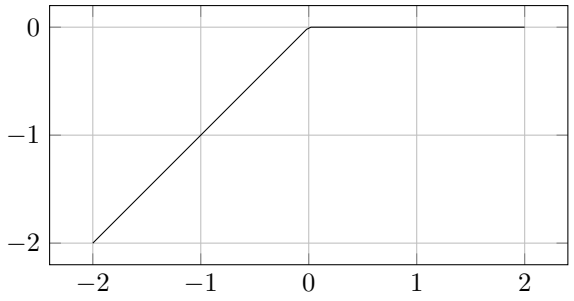


Figura 2.6: $f(x) = rampainv(x)$

Agora, os números negativos possuem o comportamento da função identidade ($f_n(x) = x$), enquanto os números positivos precisam ser subtraídos do seu módulo ($f_p(x) = x - |x|$), como mostra a figura 2.7. A solução final segue a mesma lógica do desenvolvimento da função `rampa` ($f(x) = \frac{x-|x|}{2}$).

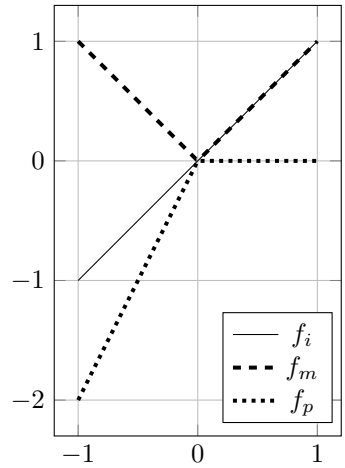


Figura 2.7: $f_i(x) = x$, $f_m(x) = |x|$, $f_p(x) = f_i(x) - f_m(x)$.

O programa 21 mostra uma implementação para a função `rampainv`.

Programa 21: rampainv.c

```
1 float rampainv(float x) {
2   return (x-fabs(x))/2;
3 }
```

Teste a implementação do programa 21 com o procedimento `plot2d`.

É mais fácil implementar a função `rampainv` após conhecer a solução da função `rampa`. Os problemas nem sempre precisam ser resolvidos do zero. Você pode e deve usar as ideias de soluções anteriores para resolver os problemas novos.

Quando o problema parece ser muito complexo e não conseguimos dividi-lo e nem conhecemos a solução de outro problema parecido, podemos criar problemas parecidos que parecem ter solução mais fácil. Daí, alteramos a solução gradativamente até resolver o problema desejado.

2.4.3 Blocos de construção

Muitas vezes, além de usar ideias inspiradas em soluções de problemas parecidos, podemos usar a solução de outro problema como parte da solução do novo. Veja que a função da figura 2.8 é muito parecida com a função da figura 2.6. Neste caso, podemos usar a própria função `rampainv` como um bloco de construção para implementar a função da figura 2.8.

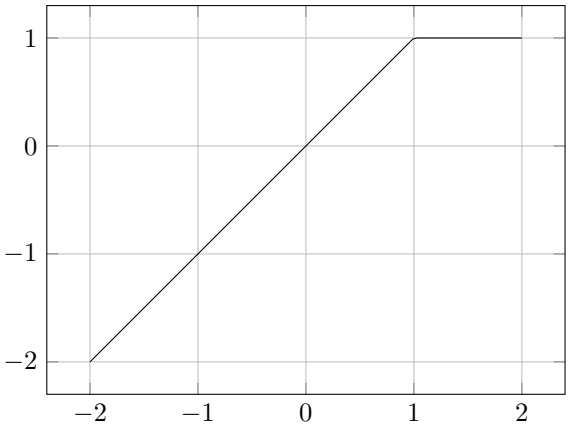


Figura 2.8: $f(x) = rampainvmax1(x)$

A função também pode ser escrita como $f(x) = \frac{(x-1)-|(x-1)|}{2} + 1$, conforme mostra a equação 2.1.

$$\begin{aligned} g(x) &= \frac{x - |x|}{2} \\ \\ f(x) &= g(x-1) + 1 \\ &= \frac{(x-1) - |(x-1)|}{2} + 1 \end{aligned} \tag{2.1}$$

Considere que $g(x)$ representa a função `rampainv`. Esta função possui a identidade para valores negativos e zero para valores positivos. Neste momento podemos usar uma técnica conhecida como abstração. Com esta técnica ignoramos a implementação da função `rampainv` e focamos nossa atenção em seu comportamento. O valor máximo que da função $g(x)$ é zero, mas queremos que o máximo seja um. Então, basta somarmos um à saída de $g(x)$, criando uma nova função $g'(x) = g(x) + 1$. O gráfico da função $g'(x)$ pode ser visto na figura 2.9. A função desejada $f(x)$ é a função $g'(x)$ deslocada para direita. Podemos deslocá-la subtraindo um de sua entrada da forma $f(x) = g'(x-1)$, que equivale a $f(x) = g(x-1) + 1$.

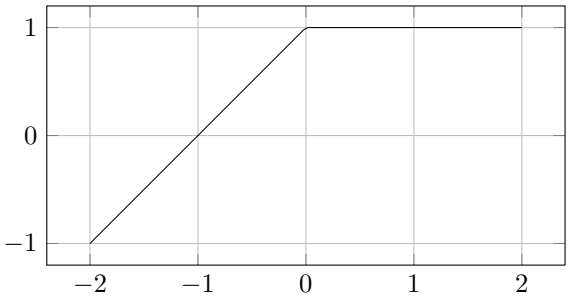


Figura 2.9: $g'(x) = g(x) + 1$

Uma possível solução é apresentada no algoritmo 22.

Programa 22: rampainvmax1.c

```
1 float rampainv(float x) {
2   return (x-fabs(x))/2;
3 }
4 float rampainvmax1(float x) {
5   return rampainv(x-1)+1;
6 }
```

Teste a implementação do programa 22 com o procedimento `plot2d`.

2.4.4 Construindo blocos

Muitas vezes ao programar, não existirão os blocos necessários para resolver nossos problemas. Quando isso ocorrer, precisamos criar nossos próprios blocos usando as técnicas apresentadas anteriormente. Suponha que você precise implementar a função da figura 2.10. Esta função que retorna um para valores maior que zero e zero para demais valores.

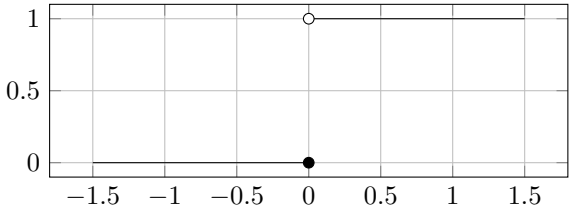


Figura 2.10: Função que retorna um para valores maior que zero e zero para demais valores.

O primeiro bloco que implementaremos é a função da figura 2.11, usando as funções `rampa` e `rampinv`. Vamos nos basear na implementação da função `rampainvmax1` apresentada no programa 22.

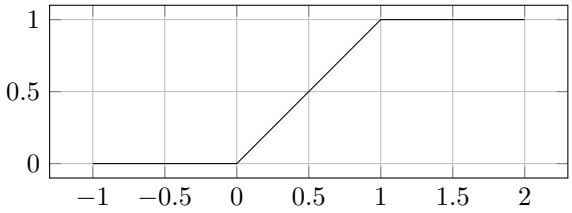


Figura 2.11: $f(x) = rampainvmax1min0(x)$

Em linguagem C, assim como a maior parte das linguagens de programação existentes, pode-se colocar o retorno de uma função diretamente como parâmetro de outra. A função do programa 23 é um pouco mais complicada do que as anteriores. Vamos dividi-la em duas partes, que chamaremos de $g(x)$ e $h(x)$. A primeira, que parte representará a função rampa, definiremos como $g(x) = \frac{x+|x|}{2}$. A segunda parte, que representará a função rampa invertida, definiremos como $h(x) = \frac{x-|x|}{2}$. O domínio de $g(x)$ é a imagem de de $h(x-1)+1$. Então, $f(x) = g(h(x-1)+1)$. A função expandida é apresentada na equação 2.2. Note que é muito mais fácil de se entender a função composta, usando a técnica de abstração, do que a função expandida.

$$
\begin{aligned}
g(x) &= \frac{x+|x|}{2} \\
h(x) &= \frac{x-|x|}{2} \\
f(x) &= g(h(x-1)+1) \\
&= g(\frac{x-1-|x-1|}{2}+1) \\
&= \frac{\frac{x-1-|x-1|}{2}+1+|\frac{x-1-|x-1|}{2}+1|}{2}
\end{aligned}
\tag{2.2}
$$

O programa 23 mostra a implementação da função $f(x)$ em C.

Assim como na equação 2.2, é muito mais fácil implementar a função `rampainvmax1min0` utilizando as funções `rampa` e `rampainv` do que implementá-la diretamente.

A função `rampainvmax1min0` resolve parcialmente o problema, como pode ser visto na figura 2.12, onde as linhas tracejadas representam a função da figura 2.10 e as linhas pontilhadas representam a função da figura 2.11.

A maior diferença entre as funções acontece entre zero e um. Uma forma de diminuir essa diferença é reduzir sua área do gráfico, fazendo uma rampa mais íngreme, como mostra a figura 2.13.

Programa 23: rampainvmax1min0.c

```
1  float rampa(float x) {
2    return (x+fabs(x))/2;
3  }
4  float rampainv(float x) {
5    return (x-fabs(x))/2;
6  }
7  float rampainvmax1min0(float x) {
8    return rampa(rampainv(x-1)+1);
9  }
```

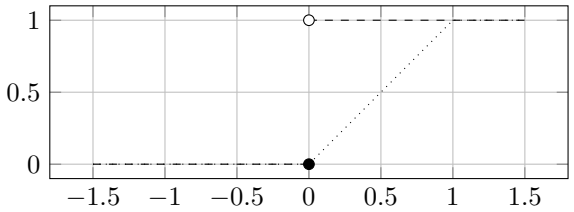


Figura 2.12: Comparação da função 2.10 com a função 2.11.

Enquanto a rampa da função na figura 2.11 tem 45° de inclinação a rampa da função na figura 2.13 tem 60° de inclinação. Vamos implementar a função da figura 2.13.

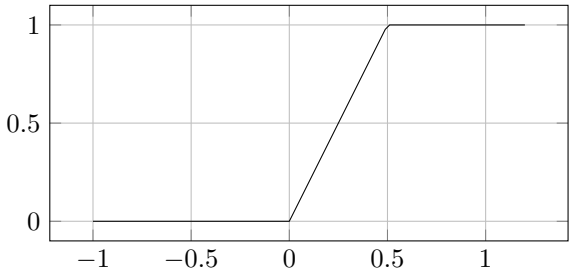


Figura 2.13: $f(x) = rampaingreme(x)$

Se nos inspirarmos na função `rampainvmax1min0` para criarmos a nova função como $f(x) = g(h(x - 0.5) + 0,5)$, a rampa terá máximo igual a $0,5$. Podemos manter o máximo igual a 1 se dividirmos o resultado por $0,5$, desta forma $f(x) = \frac{g(h(x-0.5)+0,5)}{0,5}$. A expansão desta função é apresentada na equação 2.3, onde c é a constante que determina a inclinação da rampa.

$$
\begin{aligned}
c &= 0,5 \\
g(x) &= \frac{x + |x|}{2} \\
h(x) &= \frac{x - |x|}{2} \\
f(x) &= \frac{g(h(x - c) + c)}{c} \\
&= \frac{g(\frac{x-c-|x-c|}{2} + c)}{c} \\
&= \frac{\frac{x-c-|x-c|}{2} + c + |\frac{x-c-|x-c|}{2} + c|}{2} \\
&\qquad c
\end{aligned}
\tag{2.3}
$$

O programa 24 mostra a implementação da função $f(x)$ da equação 2.3.

Programa 24: rampaingreme.c

```
1  float rampa(float x) {
2    return (x+fabs(x))/2;
3  }
4  float rampainv(float x) {
5    return (x-fabs(x))/2;
6  }
7  float rampaingreme(float x) {
8    const float c = 0.5;
9    return rampa(rampainv(x-c)+c)/c;
10 }
```

A linha 8 do programa 24 possui a palavra reservada `const`. Esta palavra indica que `aproxim` é uma constante do tipo `float`. Uma constante nada mais é do que uma variável que não pode ter o valor modificado No caso do algoritmo 24 a palavra `const` não faz muita diferença, já que a constante `c` só é utilizada dentro da pequena função `rampaingreme`. Porém, em programas grandes, a palavra `const` garante que o valor de uma constante não será alterado. Observe que quanto menor o valor da constante `aproxim`, mais íngreme é a rampa, podendo chegar a quase 90°, para valores muito próximos de zero. É isso o que a próxima função faz. Ela pode ser usada para fazer vários algoritmos importantes na programação de computadores.

Baseado na função `rampaingreme` do programa 24, podemos implementar a função `degrau`, mostrada mostrada com linhas pontilhadas na figura 2.14. Veja a comparação com a função desejada na mesma figura com linhas tracejadas.

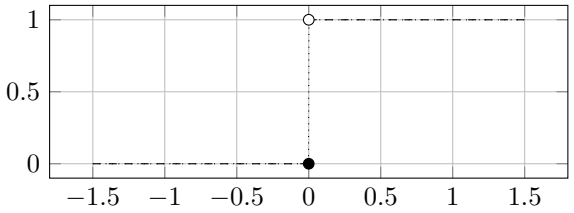


Figura 2.14: $f(x) = degrau(x)$

Computadores não possuem precisão infinita. Sempre será necessário algum arredondamento. Então, sempre teremos que definir qual precisão é necessária para determinado problema, sabendo-se que quanto mais preciso o programa, mais recursos computacionais serão necessários. Como este é um exemplo didático, vamos definir a precisão como 1/1000.

A equação da função degrau, análoga a da equação 2.3, é apresentada na equação 2.4.

$$\begin{aligned} c &= 0.0001 \\ g(x) &= \frac{x + |x|}{2} \\ h(x) &= \frac{x - |x|}{2} \\ f(x) &= \frac{g(h(x-c)+c)}{c} \\ &= \frac{g(\frac{x-c-|x-c|}{2}+c)}{c} \\ &= \frac{\frac{\frac{x-c-|x-c|}{2}+c+|\frac{x-c-|x-c|}{2}+c|}{2}}{c} \end{aligned} \tag{2.4}$$

A função é degrau é muito útil para resolver problemas. Os exercícios a seguir mostram alguns usos dela.

Programa 25: degrau.c

```
1  float rampa(float x) {
2    return (x+fabs(x))/2;
3  }
4  float rampainv(float x) {
5    return (x-fabs(x))/2;
6  }
7  float degrau(float x) {
8    const float c = 0.0001;
9    return rampa(rampainv(x-c)+c)/c;
10 }
```

2.4.5 Exercícios resolvidos

Tente resolver os exercícios sem olhar a correção comentada. Caso não consiga resolver sozinho, veja a solução apresentada e depois tente resolver novamente sem olhar a solução. As soluções estão na página 33 e as explicações estão na página 32.

1. Utilizando a função degrau apresentada no algoritmo 25, implemente uma função que receba dois valores reais e retorne 1 se o primeiro for maior do que o segundo e 0 (zero) caso contrário.

2. Utilizando a função anterior, implemente uma função que receba dois valores reais e retorne 1 se eles forem diferentes e 0 caso contrário.

3. Utilizando as duas funções anteriores, implemente uma função que receba dois valores reais e retorne 1 se eles forem iguais e 0 caso contrário.

4. Utilizando funções anteriores, implemente uma função que receba dois valores reais e retorne o maior deles. A função deve estar preparada para receber valores iguais.

5. Utilizando a função anterior, implemente uma função que receba três números reais e retorne o maior deles.

6. Implemente uma função que receba dois valores reais e retorne o menor deles. A função deve estar preparada para receber valores iguais.

7. Implemente um procedimento que receba dois números reais e os imprima em ordem decrescente.

Respostas

A seguir a explicação das soluções apresentadas no algoritmo 26

O exercício 1 é resolvido com a função `maior`. Quando a diferença entre o primeiro e o segundo parâmetro é maior do que zero a resposta é 1. Portanto, se `x` e `y` forem iguais, a diferença entre eles é zero. O valor zero é então passado para função `degrau`, que por sua vez retorna zero quando sua entrada é zero. Para valores entre zero e 10^{-4} a função `degrau` retorna o valor de entrada. Então, neste intervalo, a função maior vai falhar. Quando `x` for pelo menos 10^{-4} maior do que `y`, a função `degrau` retornará 1.

O exercício 2 usa a função `maior`. Se os valores são iguais, a função `maior` retorna zero tanto para `maior(x,y)` quanto para `maior(y,x)`. Então a soma dessas duas expressão resultam em zero, indicando que estes números não são diferentes. Por outro lado, se a expressão `maior(x,y)` resultar em zero, a expressão `maior(y,x)` necessariamente resultará em 1, e vice versa. Consequentemente, o valor máximo que a função retorna é 1, quando os valores `x` e `y` forem diferentes.

O exercício 3 duas funções criadas anteriormente, `maior` e `diferente`. Quando os valores de `x` e `y` são diferentes a função `diferente` retorna 1. Com os valores 1 e 1, a função `maior` retorna zero, indicando que os valores não são iguais. Mas quando a função diferente retorna zero, a função `maior` com os valores 1 e zero retorna 1, indicando que os valores são iguais. Em outras palavras, os números são diferentes se eles não forem iguais.

O exercício 4 é resolvido com a função `maior2`. Quando o primeiro valor é maior que o segundo `maior(x,y)` retorna 1 e `maior(y,x)` retorna 0. Então o valor de x é multiplicado por 1, o valor de `y` é multiplicado por 0. Como os valores não são iguais `x` é multiplicado por 0 na terceira parcela da soma. O

Programa 26: exer_func_degrau.c

```
1  #include <math.h>
2  float rampa(float x) {
3    return (x+fabs(x))/2;
4  }
5  float rampainv(float x) {
6    return (x-fabs(x))/2;
7  }
8  float degrau(float x) {
9    const float c = 0.0001;
10   return rampa(rampainv(x-c)+c)/c;
11 }
12 /* Exercicio 1 */
13 float maior(float x, float y) {
14   return degrau(x-y);
15 }
16 /* Exercicio 2 */
17 float diferente(float x, float y) {
18   return maior(x,y)+maior(y,x);
19 }
20 /* Exercicio 3 */
21 float igual(float x, float y) {
22   return maior(1,diferente(x,y));
23 }
24 /* Exercicio 4 */
25 float maior2(float x, float y){
26   return x*maior(x,y)+y*maior(y,x)+x*igual(x,y);
27 }
28 /* Exercicio 5 */
29 float maior3(float x1, float x2, float x3) {
30   return maior2(maior2(x1,x2),x3);
31 }
32 /* Exercicio 6 */
33 float menor2(float x, float y){
34   return x*maior(y,x)+y*maior(x,y)+x*igual(x,y);
35 }
36 /* Exercicio 7 */
37 void decrescente2(float x1, float x2) {
38   printf("%f %f\n", maior2(x1, x2), menor2(x1, x2));
39 }
```

resultado final é o valor de `x`. O contrário acontece quando o valor de `y` é maior do que `x`. O valor de `x` é multiplicado por 0 e o valor de `y` é multiplicado por 1. Novamente, os valores não são iguais e `x` é multiplicado por 0 na terceira parcela da soma. Então, o resultado final é o valor de `y`. Porém, quando os valores de `x` e `y` são iguais, tanto `maior(x,y)` quanto `maior(y,x)` retornam 0. Neste caso, `x` (que possui valor igual a `y`) é multiplicado por 1, resultando em `x`.

O exercício 5 é resolvido com a função `maior3`. A função `maior2` é usada para encontrar o maior valor entre os dois primeiros. O resultado desta chamada é usado como primeiro parâmetro de uma nova chamada da função `maior2` para compará-lo com o terceiro valor, garantindo-se assim que o valor retornado será o maior dos três. Este processo pode ser repetido para encontrar o maior de quatro, cinco, seis valores e assim por diante.

O exercício 6 é resolvido com a função `menor2`. Usa a mesma lógica da função `maior2`. Mas o valor `x` é multiplicado por `maior(y,x)` e `y` multiplicado por `maior(x,y)`.

O exercício 7 é resolvido com a função `decrescente2`. Usa as funções `maior2` e `menor2` para ordenar os valores.

2.5 Operadores relacionais

Os exercícios da subseção 2.4.5 mostram que utilizar a função `degrau` para trabalhar com funções por partes pode ser uma tarefa árdua. Para facilitar a vida dos programadores, a maioria das linguagens de programação, incluindo C, oferecem operadores lógicos. O programa 27 mostra como a função degrau pode ser implementada usando o operador lógico > (maior) ao invés das funções `rampa` e `rampainv`.

Programa 27: degrau_relacional.c

```
1 float degrau(float x) {
2   return x>0;
3 }
```

A função degrau ficou tão simples que é melhor usar o operador > diretamente do que usar a função degrau. Além disso, o resultado da função `degrau` usando o operador relacional é exato, enquanto o a função `degrau` que usa as funções `rampa` e `rampainv`, são aproximados. Assim como não faz sentido usar a função `degrau`, também não faz sentido usar a função `maior`, uma vez que o operador > consegue o mesmo resultado com muito menos código. A tabela 2.1 mostra os operadores relacionais da linguagem C.

Operador	Exemplo	Descrição
`==`	`x==y`	Verifica se `x` e `y` possuem valores iguais.
`!=`	`x!=y`	Verifica se `x` e `y` possuem valores diferentes.
`>`	`x>y`	Verifica se o valor de `x` é maior do que o valor de `y`.
`<`	`x<y`	Verifica se o valor de `x` é menor do que o valor de `y`.
`>=`	`x>=y`	Verifica se o valor de `x` é maior ou igual do que o valor de `y`.
`<=`	`x<=y`	Verifica se o valor de `x` é menor ou igual do que o valor de `y`.

Tabela 2.1: Operadores relacionais.

Os operadores relacionais em C retornam `TRUE` (verdadeiro) ou `FALSE` (falso), de acordo com a expressão. Exemplo: Uma variável `x` que possui valor três é usada na expressão `x>2`, como valor de `x` é maior do que dois, o resultado da expressão é `TRUE` (verdadeiro). Por outro lado, uma variável `y` que possui valor -1 é usada na expressão `y>0`, como o valor de `y` não é maior que zero, o resultado da expressão é `FALSE` (falso). Em C, as constantes `TRUE` e `FALSE` são representadas pelos números 1 e 0, respectivamente. Portanto, operações aritméticas são permitidas com o resultado de expressões relacionais. O programa 28 mostra como fica a função `maior2` implementada usando operadores lógicos.

Programa 28: maior2.c

```
1 float maior2(float x, float y) {
2   return x*(x>y) + y*(y>x) + x*(x==y);
3 }
```

Observe que a implementação da função `maior2` no programa 28 pode ser feita apenas substituindo a função `maior` pelo operador > e a função `igual` pelo operador ==. Então, pode-se implementar, usando apenas operadores aritméticos e relacionais, um procedimento que receba dois números reais e os imprima em ordem crescente. Uma possível implementação é apresentada no programa 29.

Programa 29: crescente2.c

```
1 void crescente2(float x1, float x2) {
2   float menor = x1*(x1<=x2)+x2*(x1>x2);
3   float maior = x1*(x1>=x2)+x2*(x1<x2);
4   printf("%f %f\n", menor, maior);
5 }
```

Exercícios

1. Implemente uma função que receba 3 números reais positivos e retorne 1 se for possível que eles sejam comprimentos de lados um triângulo e 0 se não for possível. Cada lado de um triângulo é menor do que a soma dos outros dois.

2. Implemente uma função que receba 3 números representando comprimentos de lados de um triângulo. Se for possível que esses comprimentos formem um triângulo, retorne a área desse triângulo, senão, retorne 0. A área de qualquer triângulo pode ser calculada através da raiz quadrada de t*(t-l1)(t-l2)(t-l3), onde t = (l1+l2+l3)/2 e l1, l2 e l3 são os comprimentos dos lados do triângulo.

3. Implemente uma função que receba três números reais e retorne o maior deles, usando operadores relacionais.

4. Implemente uma função que receba três números reais e retorne o menor deles, usando operadores relacionais.

5. Implemente uma função que receba três números reais e retorne o valor do meio, usando operadores relacionais.

2.6 Passagem de parâmetros por referência

A implementação do procedimento `crescente2` utilizando operadores relacionais é bem mais simples do que usando a função degrau. Mas, uma limitação muito grande do procedimento `crescente2`, do programa 29, é a necessidade de se imprimir o resultado dentro da função. E se precisarmos de usar essa função em outro tipo de interface que não um terminal, por exemplo, interfaces gráficas, páginas web ou aplicativos de celular? Esse procedimento seria inútil para tais interfaces. Outra limitação deste procedimento é a impossibilidade de ser reutilizado por outros subprogramas. Observe como o procedimento `crescente3` ficou muito mais complexo do que o procedimento `crescente2`.

Programa 30: crescente3.c

```
1  void crescente3(float x1, float x2, float x3) {
2    float menor = x1*(x1==x2)*(x1==x3)+
3                  x1*(x1<x2)*(x1<x3)+x1*(x1<x2)*(x1==x3)+x1*(x1==x2)*(x1<x3)+
4                  x2*(x2<x1)*(x2<x3)+x2*(x2<x1)*(x2==x3)+x3*(x3<x1)*(x3<x2);
5    float maior = x1*(x1==x2)*(x1==x3)+
6                  x1*(x1>x2)*(x1>x3)+x1*(x1>x2)*(x1==x3)+x1*(x1==x2)*(x1>x3)+
7                  x2*(x2>x1)*(x2>x3)+x2*(x2>x1)*(x2==x3)+x3*(x3>x1)*(x3>x2);
8    float medio = x1*(x1==x2)*(x1==x3)+
9                  x1*(x1>x2)*(x1<x3)+x1*(x1<x2)*(x1>x3)+x2*(x2>x1)*(x2<x3)+
10                 x2*(x2<x1)*(x2>x3)+x3*(x3>x1)*(x3<x2)+x3*(x3<x1)*(x3>x2)+
11                 x1*(x1==x2)*(x1<x3)+x1*(x1==x3)*(x1<x2)+x1*(x1==x2)*(x1>x3)+
12                 x1*(x1==x3)*(x1>x2)+x2*(x2==x3)*(x3<x2)+x2*(x2==x3)*(x3>x2);
13   printf("%f %f %f\n", menor, medio, maior);
14 }
```

O programa 30 mostra que esta tarefa ficou muito mais complexa do que o procedimento que imprime dois valores em ordem crescente. Podemos imaginar quão complexo seria se tentássemos implementar, usando apenas os recursos apresentados até aqui, um procedimento que ordenasse quatro, cinco, dez variáveis. É fácil notar que tais procedimentos seriam inviáveis de se programar. Seria muito mais genérico e útil um procedimento que alterasse as variáveis de entrada, colocando os valores na ordem desejada. Ou seja, o procedimento receberia dois valores por parâmetro e colocaria o menor valor no primeiro parâmetro e o maior no segundo. O programa 31 mostra uma ideia que NÃO FUNCIONA.

Onde está o erro neste programa? Será que o cálculo do menor e do maior estão errados? Na verdade, se colocarmos um `printf` dentro do procedimento, as variáveis `x1` e `x2` estarão com os valores desejados ao final de sua execução. Porém, estes valores não passam para a função `main`. Isso porque a linguagem C só conhece passagem de parâmetros por valor. Ou seja, os parâmetros `x1` e `x2` do procedimento não compartilham o mesmo espaço de memória que as variáveis `x1` e `x2` da função `main`.

Programa 31: crescente2errado.c

```
1  #include <stdio.h>
2  void crescente2errado(float x1, float x2) {
3    float menor = x1*(x1<=x2)+x2*(x1>x2);
4    float maior = x1*(x1>=x2)+x2*(x1<x2);
5    x1 = menor;
6    x2 = maior;
7  }
8  int main(){
9    float x1=3, x2=2;
10   printf("x1=%f x2=%f\n", x1, x2);
11   crescente2errado(x1,x2);
12   printf("x1=%f x2=%f\n", x1, x2);
13   return 0;
14 }
```

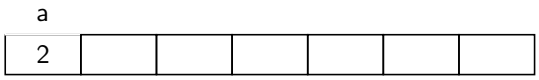
Entretanto, é possível em C, alterar valores que existem fora de um procedimento, usando a técnica de passagem de parâmetros por referência. Esta técnica consiste em passar como parâmetro o endereço de memória das variáveis que serão alteradas. Um exemplo de como se usar a passagem de parâmetros por referência é mostrado no programa 32. O procedimento `atribui10` coloca o valor 10 à posição de memória passada por referência.

Programa 32: atribui10.c

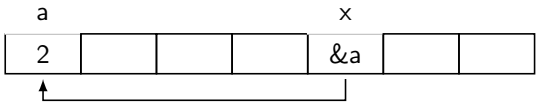
```
1  #include <stdio.h>
2  void atribui10(float *x) {
3    *x = 10;
4  }
5  int main() {
6    float a=2;
7    printf("a=%f\n", a);
8    atribui10(&a);
9    printf("a=%f\n", a);
10   return 0;
11 }
```

Para declarar um parâmetro que recebe um endereço de memória ao invés de um valor usa-se a expressão `tipo *variavel`, como se vê na linha 2 do algoritmo. Quando declaramos `float *x`, o parâmetro `x` não aceita valores do tipo `float`. Ele só aceita endereços de memória reservados para variáveis do tipo `float`. É importante definir o tipo para que o SO saiba o tamanho do endereço de memória que foi passado por parâmetro. Variáveis que recebem posições de memória são normalmente chamadas de ponteiros. Dentro do procedimento `atribui10`, o parâmetro `x` terá uma cópia do endereço da variável externa `a`. Também pode-se dizer que o parâmetro `x` aponta para o endereço de memória da variável `a`. Por isso o nome ponteiro. Através deste endereço, o procedimento consegue alterar o valor da variável externa usando a expressão `*x = 10;` na linha 3. A expressão `*ponteiro` retorna o conteúdo de um endereço de memória. Como o procedimento `atribui10` só aceita endereços de memória de variáveis do tipo `float`, a chamada deste procedimento deve passar o endereço da variável, não a variável diretamente. Isso pode ser feito como a expressão usada na linha 8: `atribui10(&a);`. O operador `&` retorna o endereço da variável. A linha 9 mostra que o valor da variável `a` foi alterado de 2 para 10.

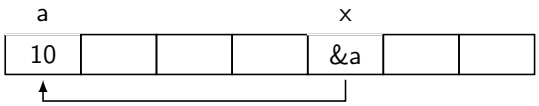
A figura 2.15 ilustra a memória do computador durante a execução do programa 32. Quando o programa se inicia, o SO reserva uma quantidade de memória, que está inicialmente livre. Na linha 6, o programa define uma posição de memória para guardar o valor da variável `a` e atribui o valor 2 a ela (figura 2.15a). A linha 8 chama o procedimento `atribui10`, criando a variável do tipo ponteiro para float `x` (`float*`), atribuindo a ela o endereço da variável `a` (`&a`) (figura 2.15b). A linha 3 altera o valor da posição de memória apontada por `x`, fazendo com que o valor de `a` mude para 10 (figura 2.15c).



(a) Alocação e atribuição do valor 2 à variável **a** (linha 6)



(b) Alocação do atributo ***x** e atribuição do endereço da variável **a** a ele. (linha 8 e 2)



(c) Atribuição do valor 10 à posição de memória apontada por **x** (linha 3)

Figura 2.15: Memória do computador durante a execução do algoritmo 32.

Veja como seria um procedimento que recebe dois parâmetros por referência e troca seus valores nas variáveis externas.

Programa 33: troca.c

```
1  #include <stdio.h>
2  void troca(float *x, float *y) {
3     float aux = *x;
4     *x = *y;
5     *y = aux;
6  }
7  int main() {
8     float a=1, b=2;
9     printf("a=%f b=%f\n", a, b);
10    troca(&a, &b);
11    printf("a=%f b=%f\n", a, b);
12    return 0;
13 }
```

Vamos analisar o programa 33 em sua ordem de execução. Na linha 8 as variáveis `a` e `b` são criadas e valores são atribuídos a elas. A linha 9 apenas imprime as variáveis para sabermos a ordem em que estão. A linha 10 passa o endereço de memória das variáveis `a` e `b` para o procedimento `troca`. O procedimento `troca` define na linha 2 que serão recebidos ponteiros ao invés de valores. Ele usa uma variável auxiliar `aux` para guardar temporariamente o valor contido na posição de memória apontada por `x`, na linha 3. Vale a pena ressaltar que a variável `aux` não é um ponteiro. É uma variável do tipo `float` que guarda o valor contido na posição de memória apontada pela variável ponteiro `x`. Na linha 4, a posição de memória apontada por `x` recebe o valor contido na posição de memória apontada por `y`. Neste momento, as posições de memória apontadas por `x` e por `y` possuem o mesmo valor, mas estão em locais diferentes na memória principal. Na linha 5, o valor de `aux`, que era o valor contido em `x`, é agora atribuído à posição de memória apontada por `y`. Finalizando assim, a troca dos valores das posições de memória referenciadas. A verificação dessa troca pode ser observada na linha 11, e então, o programa é finalizado.

2.6.1 Exercícios Resolvidos

1. Inspirando-se no program 33, corrija o programa 31 de forma que ele funcione.

2. Utilizando o procedimento `crescente2correto`, implemente o procedimento `crescente3facil`, que receba três valores e os coloque em ordem crescente.

3. Utilizando o procedimento `crescente3facil`, implemente um procedimento que receba quatro valores e os coloque em ordem crescente.

2.6.2 Soluções

A função `main` do programa 34 é muito parecida com a função `main` do programa 33. Apenas os nomes das variáveis são diferentes. O procedimento `crescente2correto` muda apenas alguns detalhes em relação ao procedimento `crescente2errado`. A primeira diferença está na assinatura da função. No procedimento correto são declarados ponteiros como parâmetros. Para minimizar as alterações e melhorar a legibilidade do código, o nome dos parâmetros também foram alterados, e agora são `a` e `b`. Criando variáveis locais com o nome dos parâmetros anteriores podemos manter o código que definia o menor e o maior valor. As linha 6 e 7 atualizam os valores das posições de memória que estão nas variáveis externas.

Programa 34: crescente2correto.c

```
1  #include <stdio.h>
2  void crescente2correto(float *a, float *b) {
3    float x1 = *a, x2 = *b;
4    float menor = x1*(x1<=x2)+x2*(x1>x2);
5    float maior = x1*(x1>=x2)+x2*(x1<x2);
6    *a = menor;
7    *b = maior;
8  }
9  int main(){
10   float x1=3, x2=2;
11   printf("x1=%f x2=%f\n", x1, x2);
12   crescente2correto(&x1,&x2);
13   printf("x1=%f x2=%f\n", x1, x2);
14   return 0;
15 }
```

A maior vantagem do procedimento `crescente2correto`, entretanto, é a possibilidade de sua reutilização, como mostra o programa 35. Isso é possível graças à passagem por referência usando ponteiros.

Programa 35: crescente3facil.c

```
1  void crescente3facil(float *x1, float *x2, float *x3) {
2    crescente2correto(x1,x2);
3    crescente2correto(x2,x3);
4    crescente2correto(x1,x2);
5  }
```

Pode-se observar que o procedimento `crescente3facil` é muito mais fácil de implementar e entender do que o procedimento `crescente3` do programa 30. Além disso, o fato de não usar a saída padrão (terminal) para apresentar o resultado torna este procedimento muito mais útil e versátil. Ele pode ser usado para qualquer tipo de interface e também pode ser aproveitado por outras partes do código. Vale ressaltar que ao chamar o procedimento `crescente2correto`, nas linhas 2, 3 e 4, não se usou o operador `&`. Isso porque as variáveis `x1`, `x2` e `x3` já possuem posições de memória armazenadas, ou seja, são ponteiros, e são essas posições que precisam ser alteradas. O teste desse procedimento fica a cargo do leitor.

Programa 36: crescente4facil.c

```
1  void crescente4facil(float *x1, float *x2, float *x3, float *x4) {
2    crescente3facil(x1,x2,x3);
3    crescente3facil(x2,x3,x4);
4    crescente3facil(x1,x2,x3);
5  }
```

O entendimento do algoritmo 36 é análogo ao do algoritmo 35. Nota-se portanto, que a implementa-

ção e algoritmos mais complexos que ordene quatro, cinco ou dez variáveis, poderia ser facilmente feita implementando-se procedimentos mais simples e usando-os na implementação dos procedimentos mais complexos. O teste desse procedimento fica a cargo do leitor.

2.7 scanf

Os programas implementados até aqui não estão flexíveis. As funções e procedimentos são testadas com valores fixos. Antes da seção anterior, não sabíamos como alterar o valor de uma variável externa. Mas agora que sabemos usar a passagem de parâmetros por referência, podemos usar a função `scanf`. Esta função também faz parte da biblioteca `stdio.h`. De fato, a biblioteca `stdio.h` é a biblioteca de entrada e saída padrão, mas até aqui nós só usamos a saída com a função `printf`. O programa 37 mostra um exemplo da utilização da função `scanf`.

Programa 37: raizquadrada.c

```
1 #include <stdio.h>
2 #include <math.h>
3 int main(void) {
4   float x;
5   printf("Digite um numero positivo: ");
6   scanf("%f", &x);
7   printf("A raiz quadrada de %f eh %f\n", x, sqrt(x));
8   return 0;
9 }
```

Na linha 6 programa 37 a função `scanf` é usada. Note que a sintaxe desta função é muito parecida com a sintaxe da função `printf`. O primeiro argumento é uma cadeia de caracteres que usa caracteres especiais para determinar o tipo de dado que será lido. Assim como a função `printf`, os caracteres `%f` e `%d` são usados para variáveis do tipo `float` e `int`, respectivamente. Como o valor digitado no terminal deve ser armazenado em uma posição de memória, é necessário reservar um espaço de memória para guardar este valor. Isso é feito através da declaração da variável `x` na linha 4. A endereço dessa variável (passagem por referência) é passado para a função `scanf`, que armazena o valor digitado no terminal na posição de memória na variável `x`.

Exercícios

Use a função `scanf` para a leitura dos dados nos exercícios.

1. Implemente um programa que leia 5 valores reais e os imprima em ordem crescente.

2. Implemente um programa que leia uma temperatura em Celsius e imprima a temperatura em Fahrenheit. O cálculo de conversão deve ser feito em uma função separada.

3. Faça um programa que calcule o Índice de Massa Corpórea de uma pessoa. $IMC = \frac{m}{h^2}$, onde m é a massa da pessoa em Kg e h é a altura em centímetros.

4. Implemente um programa que leia dois números positivos representando os lados de um retângulo e imprima sua área e seu perímetro. Os cálculos da área e do perímetro devem ser feitos em funções separadas.

5. Implemente um programa que leia um valor positivo representado o raio de um círculo e e imprima sua área e seu perímetro. Os cálculos da área e do perímetro devem ser feitos em funções separadas.

6. Implemente um programa que leia dois valores representando os catetos de um triângulo retângulo e imprima os ângulos que este triângulo tem.

Considerações do capítulo

- Neste capítulo vimos uma técnica de modularização de algoritmos através de subprogramas, que podem ser funções ou procedimentos.
  - Funções sempre retornam um valor e não devem ter efeitos colaterais no algoritmo.
  - Procedimentos não devem retornar um valor e podem ou não alterar o estado dos programas, como mostrado na seção 2.6.
- Subprogramas podem ser utilizados por outros subprogramas.
- A implementação de um subprograma deve sempre visar sua reutilização.
- Subprogramas devem ser mais concisos e genéricos o possível.
- É possível resolver uma infinidade de problemas usando apenas operadores aritméticos. Porém, outros tipos de operadores, como operadores relacionais, podem facilitar bastante a implementação das soluções.
- Passagem de parâmetros por referência, em linguagem C, só pode ser feita através de ponteiros.
- As funções `printf` e `scanf`, da biblioteca `stdio.h`, são usadas para saída e entrada padrão em C.

Capítulo 3

Condicionais

Dividindo programas em funções e procedimentos podemos implementar soluções para resolver infinitos tipos de problemas. Porém, as linguagens de programação geralmente possuem recursos que facilitam a implementação das soluções. Um desses recursos são os condicionais.

3.1 O comando `if`

Para entender melhor a utilidade dos condicionais, vamos implementar um programa que pede para o usuário digitar um número e diz se este número é ímpar. Implementar esse programa seria muito complicado se não existisse o comando condicional `if`. A sintaxe do comando `if` é:

```
if(/* condicao */) {
 /* Bloco de comandos */
}
```

Para entender melhor como este comando funciona, veja o programa 38.

Programa 38: impar.c

```
1 #include <stdio.h>
2  int main(void) {
3    int n;
4    printf("Digite um numero inteiro: ");
5    scanf("%d", &n);
6    if(n%2) {
7      printf("%d eh impar.\n", n);
8    }
9  }
```

Este programa lê um número inteiro pelo terminal e o armazena na variável `n` (linhas 3, 4 e 5). Na linha 6, dentro dos parênteses do comando `if`, é calculado o resto da divisão inteira de `n` por 2 usando o operador %. A resposta da expressão `n%2` será 1 se o resto da divisão inteira de `n` por 2 for 1. E retornará 0 caso o resto da divisão inteira de `n` por 2 for de zero. Em outras palavras, a expressão diz se o valor em `n` é ímpar ou não. O comando `if` executa o bloco de comandos dentro das chaves "{"`comando;`"}" se o valor dentro dos parênteses for diferente de zero. Considere que o valor digitado seja 7. O resto da divisão inteira de 7 por 2 é 1 (7%2==1). Então, o programa imprimirá "`7 eh impar.`" na tela. Por outro lado, se o valor digitado for 10, por exemplo, o programa não imprimirá nada além de "`Digite um numero inteiro:` ".

Considere fazer um programa que identifique se um valor digitado é par. Este programa precisaria verificar se o resto da divisão inteira do número digitado é 0. Podemos implementar isso usando o operador relacional `==`, como mostra o programa 39. Este programa lê um número inteiro pelo terminal e o armazena na variável `n` (linhas 3, 4 e 5), assim como o programa anterior. Na linha 6, dentro dos parênteses do comando `if`, é calculado o resto da divisão inteira de `n` por 2 usando o operador %. O resultado dessa operação agora é comparado ao valor zero usando o operador relacional `==`. Como já vimos no capítulo anterior, operadores

Programa 39: par.c

```
1 #include <stdio.h>
2 int main(void) {
3   int n;
4   printf("Digite um numero inteiro: ");
5   scanf("%d", &n);
6   if(n%2 == 0) {
7     printf("%d eh par.\n", n);
8   }
9 }
```

relacionais retornam valores 0 ou 1. Nesta nova situação, o bloco de comandos dentro das chaves é executado se o resto da divisão inteira de `n` por 2 for igual a 0.

Usando apenas o condicional `if`, podemos implementar um programa que leia um número pelo terminal e imprima se ele é par ou ímpar.

Programa 40: parouimpar.c

```
1 #include <stdio.h>
2 int main(void) {
3   int n;
4   printf("Digite um numero inteiro: ");
5   scanf("%d", &n);
6   if(n%2) {
7     printf("%d eh impar.\n", n);
8   }
9   if(n%2 == 0) {
10    printf("%d eh par.\n", n);
11  }
12 }
```

O programa 40 resolve o problema, mas este programa fica mais simples quando resolvido com o comando `else`, que será apresentado na seção 3.2.

3.2 O comando `else`

Nota-se que no programa 40 o comando da linha 6 é o contrário do comando da linha 9. Para estes casos, podemos usar o comando `else`. O comando `else` só pode ser usado com um comando `if`. A sintaxe do comando `if` com `else` é:

```
if(/* condicao */) {
 /* Bloco de comandos para condicao verdadeira */
} else {
 /* Bloco de comandos para condicao falsa */
}
```

Para entender melhor como este comando funciona, veja o programa 41.

O comando `else` é muito útil para casos como o apresentado no programa 41, onde é verificado na linha 6 se o número é ímpar. Se o número for ímpar, o bloco de comandos do `if` é executado, imprimindo a mensagem que diz que o número é ímpar. Caso o número digitado não seja ímpar, ele só pode ser par. Então, nenhuma verificação adicional precisa ser feita. Executa-se o bloco de comandos do `else` diretamente.

Programa 41: parouimparelse.c

```
1  #include <stdio.h>
2  int main(void) {
3    int n;
4    printf("Digite um numero inteiro: ");
5    scanf("%d", &n);
6    if(n%2) {
7      printf("%d eh impar.\n", n);
8    } else {
9      printf("%d eh par.\n", n);
10   }
11 }
```

3.3 Condicionais aninhados

Os blocos de comandos dentro de `if` e `else` podem conter outros comandos de condição. Isso é chamado de estrutura aninhada, por lembrar um ninho e suas camadas. Veja o exemplo do programa 42.

Programa 42: positivo_ou_negativo.c

```
1  #include <stdio.h>
2  int main(void) {
3    float n;
4    printf("Digite um numero real: ");
5    scanf("%f", &n);
6    if(n>0) {
7      printf("%f eh positivo.\n", n);
8    } else {
9      if(n<0) {
10       printf("%f eh negativo.\n", n);
11     } else {
12       printf("%f eh neutro.\n", n);
13     }
14   }
15 }
```

As estruturas aninhadas podem estar tanto dentro do bloco do `if` quanto do bloco do `else`. O programa 42 exige uma estrutura aninhada porque não é suficiente saber que um número é negativo se ele não for positivo. Existe o número zero que não está em nenhuma dessas categorias.

3.4 Subprogramas com condicionais

Condicionais também podem ser usados em subprogramas. Por exemplo, o programa 43 mostra um procedimento que recebe um valor e diz se ele é negativo.

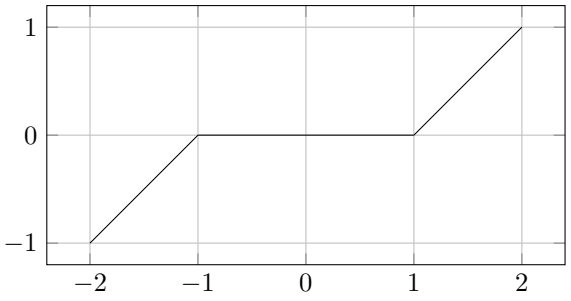
Programa 43: negativo.c

```
1  #include <stdio.h>
2  void negativo(float n){
3    if(n<0){
4      printf("%f eh negativo.\n", n);
5    }
6  }
```

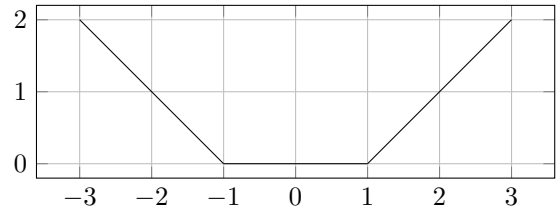
Exercícios

Para os exercícios seguintes use `if` ou `if` e `else`, aninhados ou não, conforme a necessidade:

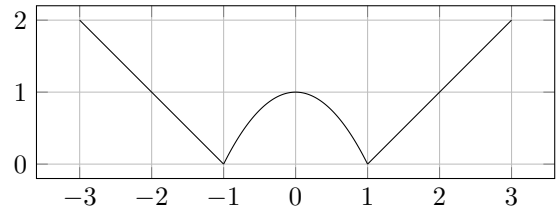
1. Implemente um procedimento que receba dois números e imprima o maior deles.
2. Implemente uma função que receba dois números e retorne o maior deles.
3. Implemente um procedimento que receba dois números e os imprima em ordem crescente.
4. Implemente um procedimento que receba dois ponteiros e coloque seus valores em ordem crescente.
5. Implemente a função apresentada na figura a seguir.



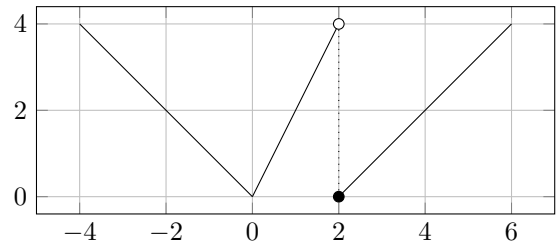
6. Implemente a função apresentada na figura a seguir.



7. Implemente a função apresentada na figura a seguir.



8. Implemente a função apresentada na figura a seguir.



9. Implemente uma função que receba 3 valores reais e retorne o maior deles.
10. Implemente uma função que receba 3 valores reais e retorne o do meio.

3.5 Operadores Lógicos

Existem situações em que precisamos analisar condições simultaneamente. Vejamos o exemplo de uma função que recebe três valores e retorna o do meio. Podemos implementar essa função usando `if`s aninhados, como mostra o programa 44.

Programa 44: meio_aninhado.c

```
1  float meio_aninhado(float a, float b, float c){
2    if(a>b){
3      if(b>c){
4        return b;
5      }else{
6        if(a>c){
7          return c;
8        }else{
9          return a;
10       }
11     }
12   }else{
13     if(a>c){
14       return a;
15     }else{
16       if(b>c){
17         return c;
18       }else{
19         return b;
20       }
21     }
22   }
23 }
```

Apesar do programa 44 ser bem eficiente, o programa 45 é bem mais intuitivo e legível.

Programa 45: meio_logico.c

```
1  #include <stdio.h>
2  float meio_logico(float a, float b, float c){
3    if(a<=b && b<=c || a>=b && b>=c){
4        return b;
5    }
6    if(b<=a && a<=c || b>=a && a>=c){
7        return a;
8    }
9    return c;
10 }
```

O programa 45 usa o operador lógico `&&` para fazer uma conjunção (AND) de proposições lógicas. Em outras palavras, o operador `&&` precisa de dois operandos, que devem ser proposições lógicas (tipo `x>y` ou `a==b`). O operador `&&` retorna TRUE, se e somente se o resultado de seus dois operandos for TRUE. A linguagem C também possui um operador lógico para disjunção (OR), o operador `||`. Este operador também precisa de dois operandos, mas retorna TRUE se pelo menos um deles for TRUE. Outro operador é o operador `!`, usado para negação (NOT). Este possui apenas um operando e nega seu valor. Isto é, retorna TRUE quando o operando for FALSE e FALSE quando o operando for TRUE. A tabela 3.1 mostra o resultado dos operadores para todas as situações possíveis. Quando usados numa mesma expressão, o operador `&&` tem precedência em relação ao operador `||`.

A	B	A&&B	A||B	!A	!B
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

Tabela 3.1: Tabela Verdade

Exercícios

1. Implemente uma função que receba como parâmetro um número real representando o salário de uma pessoa e um número inteiro representando sua idade. Se a pessoa tiver menos de 18 ou mais de 65 e receber entre 500 e 1000, seu valor de imposto de 10% sobre salário. Salários menores que 500 não pagam imposto. As demais situações pagam 20% sobre o salário.

2. Faça um programa que leia o salário e a idade de uma pessoa e retorne o imposto devido, usando a função anterior.

3. Implemente uma função que verifique se um ano é ou não bissexto.

4. Faça um programa que leia um valor inteiro representando um ano e use a função do exercício anterior para imprimir na tela se este ano é ou não bissexto.

5. Faça um programa que leia um valor inteiro representando um ano e imprima na tela se este ano é ou não bissexto, sem utilizar a função do exercício 3.

6. Faça um programa que calcule as raízes de uma função do segundo grau, quando existir pelo menos uma.

7. Faça um programa que verifique se um triângulo é equilátero, isósceles ou escaleno.

8. Implemente um programa que verifique se três valores podem ser ângulos de um triângulo.

9. Implemente um programa que verifique se três valores podem ser lados de um triângulo.

10. Faça um programa que calcule o valor de uma conta de energia elétrica, de acordo com o consumo, conforme valores apresentados na tabela 3.2. Se a conta for mais alta que 400 ela terá uma sobretaxa de 15%. A conta mínima é de 100.

Consumo	Tarifa
kWh $\leq$ 200	1.20
200 $<$ kWh $\leq$ 400	1.50
400 $<$ kWh $\leq$ 600	1.80
600 $<$ kWh	2.00

Tabela 3.2: Tarifas por consumo.

11. Implemente um programa que leia um número inteiro representando um dia da semana e imprima o nome do dia.

12. Implemente um procedimento que receba um algarismo e imprima seu nome.
Exemplo: entrada=4, saída="quatro".

13. Implemente uma função que receba um número inteiro entre 1 e 12 e retorne o número de dias do mês correspondente ao número.

14. Implemente um programa para calcular a área de várias formas geométricas.

3.6 Recursão

Suponha que desejamos implementar um procedimento que receba um valor inteiro positivo e imprima todos entre ele e zero regressivamente. As linguagens de programação possuem uma característica conhecida como recursão, onde a implementação de uma função ou procedimento pode fazer uma auto-chamada. Veja o programa 46, onde o procedimento `contagem_regressiva` faz uma auto-chamada na linha 5, reduzindo o valor do parâmetro em 1.

Programa 46: contagem_regressiva.c

```
1  #include <stdio.h>
2  void contagem_regressiva(int n){
3     if(n>=0) {
4        printf("%d\n", n);
5        contagem_regressiva(n-1);
6     }
7  }
8  int main(void) {
9     printf("Contagem regressiva:\n");
10    contagem_regressiva(10);
11    return 0;
12 }
```

Três pontos importantes devem ser observados no procedimento `contagem_regressiva` para que a recursão funcione. A linha 3 mostra a condição para execução do procedimento. O procedimento não é executado para valores negativos. A linha 4 executa uma parte do procedimento. Este deve imprimir $n+1$ números, mas para isso, precisa executar um comando de impressão $n+1$ vezes. Como a quantidade de impressões é variável, não é possível escrever $n+1$ comandos de impressão. Então, o procedimento imprime o valor de `n` e passa os demais valores para um outro procedimento mais simples, que deve imprimir todos os demais valores, exceto `n`. Isso é feito na linha 5, onde o tamanho do problema, que é imprimir $n+1$ números é reduzido para imprimir apenas n números. O procedimento `contagem_regressiva` será chamado $n+1$ vezes e, consequentemente, imprimirá $n+1$ valores.

Podemos concluir que um subprograma recursivo sempre precisará de pelo menos um parâmetro, no qual será avaliado se o subprograma será executado ou não.

3.6.1 Ordem dos comandos

Observe o programa 47. É praticamente igual ao programa 46, exceto pela ordem das linhas 4 e 5. A troca dessas linhas faz com que os resultados dos procedimentos sejam completamente diferentes. O procedimento `contagem_regressiva` resolve parte do problema, reduz o tamanho do problema e passa o problema reduzido para outra chamada do próprio procedimento. O procedimento `contagem_progressiva` reduz o trabalho

Programa 47: contagem_progressiva.c

```
1  #include <stdio.h>
2  void contagem_progressiva(int n){
3     if(n>=0) {
4        contagem_progressiva(n-1);
5        printf("%d\n", n);
6     }
7  }
8  int main(void) {
9     printf("Contagem progressiva:\n");
10    contagem_progressiva(10);
11    return 0;
12 }
```

do problema e passa o problema reduzido para outra chamada do próprio procedimento. Só depois que o problema reduzido foi resolvido é que o procedimento resolve sua parte do problema. Alguns subprogramas podem não ser afetados pela ordem em que os comandos são colocados, mas normalmente a ordem dos comandos é importante.

3.6.2 Funções intrinsecamente recursivas

Várias funções matemáticas são definidas por recursão. O exemplo mais clássico de função recursiva é o fatorial, definido como:

$$n! = \begin{cases} 1 & \text{se } n \leq 1, \\ n \times (n-1)! & \text{caso contrário.} \end{cases}$$

A equação pode ser facilmente traduzida para linguagem C, conforme mostra o programa 48.

Programa 48: fatorial.c

```
1  #include <stdio.h>
2  int fatorial(unsigned int n){
3    if(n>1){
4      return n*fatorial(n-1);
5    }
6    return 1;
7  }
8  int main(void) {
9    printf("5! = %d\n", fatorial(5));
10   return 0;
11 }
```

A função `fatorial` no programa 48 recebe como único parâmetro um inteiro sem sinal (`unsigned int`).

3.6.3 Exercícios

Os exercícios a seguir podem ou não usar funções auxiliares.

1. Implemente uma função recursiva que calcule a soma dos n primeiros números naturais ($\mathcal{N}^*$).

2. Implemente uma função recursiva que retorne a quantidade de algarismos de um número inteiro.

3. Implemente uma função recursiva que calcule a soma dos algarismos de um número inteiro.

4. Implemente uma função recursiva que retorne o enésimo termo da sequência de Fibonacci.

5. Implemente um procedimento recursivo que imprima, a partir de um número natural maior do que zero, os números da sequência de Collatz, definida como:

$$C(x) = \begin{cases} \frac{x}{2} & \text{se } x \equiv 0 \pmod 2 \\ 3 \times x + 1 & \text{se } x \equiv 1 \pmod 2 \end{cases}$$

   A sequência termina em 1.

6. Implemente uma função recursiva que encontre o máximo divisor comum de dois números inteiros.

7. Implemente uma função recursiva que encontre o mínimo múltiplo comum de dois números inteiros.

8. Implemente uma função recursiva para verificar se um número inteiro é primo.

3.7 Laços de repetição

A maior parte dos programas recursivos podem ser implementados de forma iterativa. Veja o exemplo do programa 49. Do lado esquerdo temos o procedimento de contagem regressiva implementado de forma recursiva, e do lado direito temos outro procedimento que gera o mesmo resultado de forma iterativa, isto é, repetindo passos. O comando `while` executa o bloco de comando entre as chaves **enquanto** a condição entre parênteses for verdadeira. Observe que os dois procedimentos possuem um valor inicial, definido na linha 1, uma condição de execução do bloco, definida na linha 2, a parte do problema a ser resolvida a cada chamada de procedimento ou iteração, e um passo para reduzir o tamanho do problema, na linha 4.

Programa 49: contagem_iterativa.c

```
1 void contagem_recursiva(int n){
2   if(n>=0) {
3     printf("%d\n", n);
4     contagem_recursiva(n-1);
5   }
6 }
```

```
1 void contagem_iterativa(int n){
2   while(n>=0) {
3     printf("%d\n", n);
4     n=n-1;
5   }
6 }
```

Entretanto, nem todos programas recursivos têm sua versão iterativa tão parecida. Veja a comparação da contagem progressiva apresentada no programa 50. Ainda assim, os dois programas possuem valor inicial, condição para execução e diminuição do tamanho do problema. Porém, programas iterativos sempre devem resolver parte do problema antes de diminuir o tamanho do problema, diferentemente de programas recursivos.

Programa 50: contagem_iterativa2.c

```
1 void contagem_recursiva2(int n){
2   if(n>=0) {
3     contagem_recursiva2(n-1);
4     printf("%d\n", n);
5   }
6 }
```

```
1 void contagem_iterativa2(int n){
2   int i=0;
3   while(i<=n) {
4     printf("%d\n", i);
5     i=i+1;
6   }
7 }
```

Até programas intrinsecamente recursivos podem ter versões iterativas com algumas adaptações, como mostra o programa 51. Os dois programas possuem a mesma condição de execução e a mesma redução de problema, porém a forma de guardar cada passo é diferente. O programa recursivo delega a resolução de um problema menor para outra chamada da função antes de retornar seu resultado. O programa iterativo, resolve a parte de cada iteração antes de enviar o problema menor para iteração seguinte. Por isso este usa uma variável auxiliar `f`.

Programa 51: fatorial_iterativo.c

```
1 int fatorial_recursivo(int n){
2   if(n>1){
3     return n*fatorial_recursivo(n-1);
4   }
5   return 1;
6 }
```

```
1 int fatorial_iterativo(int n){
2   int f = 1;
3   while(n>1){
4     f = f*n;
5     n = n-1;
6   }
7   return f;
8 }
```

3.7.1 Exercícios

1. Implemente um procedimento que imprima os n primeiros números naturais ($\mathcal{N}^*$).
2. Implemente uma função que retorne a soma dos n primeiros números naturais ($\mathcal{N}^*$).
3. Implemente um procedimento que leia n números e imprima a soma e a média deles.
4. Implemente um procedimento que imprima a tabuada de um número inteiro.
5. Implemente um procedimento que imprima os n primeiros números naturais ímpares e sua soma.
6. Implemente uma função que calcule a soma $s(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$.
7. Implemente uma função que calcule a série $s(x, n) = 1 - \frac{x^2}{2} + \frac{x^3}{3} - \ldots \pm \frac{x^n}{n}$.
8. Implemente um procedimento que imprima um triângulo, com o padrão a seguir para $n = 4$.
```
*
* *
* * *
* * * *
```
9. Implemente um procedimento que imprima um triângulo, com o padrão a seguir para $n = 4$.
```
1
1 2
1 2 3
1 2 3 4
```
10. Implemente um procedimento que imprima um triângulo, com o padrão a seguir para $n = 4$.
```
1
2 2
3 3 3
4 4 4 4
```
11. Implemente um procedimento que imprima um triângulo, com o padrão a seguir para $n = 4$.
```
1
2 3
4 5 6
7 8 9 10
```
12. Implemente um procedimento que imprima um triângulo de Pascal, com o padrão a seguir para $n = 5$.
```
1 1 1 1 1
1 2 3 4
1 3 6
1 4
1
```
Dica: Use a função fatorial para calcular o Binômio de Newton para cada número do triângulo.
13. Implemente um procedimento que imprima um número inteiro invertido (12345 $\Rightarrow$ 54321).
14. Implemente uma função que verifique se um número é palíndromo ou não.
15. Implemente uma função que verifique se um número é perfeito ou não. Um número perfeito é um número natural para o qual a soma de todos os seus divisores naturais próprios (excluindo ele mesmo) é igual ao próprio número.
16. Implemente uma função que dados dois números inteiros, calcula a quantidade de números perfeitos que existem entre eles.
17. Implemente uma função que calcule quantos números primos existem entre dois números naturais.

18. Implemente uma função que verifique se um número inteiro pode ser expresso pela soma de dois números primos.

19. Implemente uma procedimento que verifique se um número inteiro pode ser expresso pela multiplicação de dois números primos e imprima estes números quando possível.

20. A proporção áurea, número de ouro, número áureo ou proporção de ouro é uma constante real algébrica irracional denotada pela letra grega ϕ (PHI), em homenagem ao escultor Phideas, que a teria utilizado para conceber o Parthenon, e com o valor arredondado a três casas decimais de 1,618. Também é chamada de seção áurea (do latim *sectio aurea*), razão áurea, razão de ouro, divina proporção, divina seção (do latim *sectio divina*), proporção em extrema razão, divisão de extrema razão ou áurea excelência. O número de ouro é ainda frequentemente chamado razão de Phidias . Como é um número extraído da sequência de Fibonacci, o número áureo representa diretamente uma constante de crescimento. O número áureo é aproximado pela divisão do enésimo termo da Série de Fibonacci $(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...,$ na qual cada número é a soma dos dois números imediatamente anteriores na própria série) pelo termo anterior. Essa divisão converge para o número áureo conforme tomamos cada vez maior. Podemos ver um exemplo dessa convergência a seguir, em que a série de Fibonacci está escrita até seu sétimo termo [1, 1, 2, 3, 5, 8, 13]:

$$\frac{2}{1} = 2, \frac{3}{2} = 1.5, \frac{5}{3} = 1.666..., \frac{8}{5} = 1.6, \frac{13}{8} = 1.625, ...$$

Escreva um programa que faça várias iterações dividindo o enésimo termo da Série de Fibonacci pelo seu antecessor, até que a diferença absoluta entre valores encontrados em iterações sucessoras seja inferior a 0.000001 e imprima o último valor encontrado.

3.7.2 Laço *for*

Usar contadores em repetições é tão comum que a linguagem C possui um comando que resume em uma linha as três partes fundamentais de uma contagem, que são, valor inicial, condição de execução e passo (incremento ou decremento). O programa 1 mostra uma comparação dos comandos `while` e `for`.

Programa 52: contagem_for.c

```
1 void contagem_while(int n){
2   int i=0;
3   while(i<=n) {
4       printf("%d\n", i);
5       i=i+1;
6   }
7 }
```

```
1 void contagem_for(int n){
2   int i;
3   for(i=0; i<=n; i=i+1) {
4       printf("%d\n", i);
5   }
6 }
```

Qualquer código feito com o comando `for` também pode ser feito com o comando `while`. Entretanto, o comando `for` é mais agradável para programar e identificar contagens por ter suas três partes dispostas em uma só linha de código.

3.7.3 Exercícios

Para exercitar o comando `for`, reimplemente alguns exercícios feitos com o comando `while`.

Capítulo 4

Estruturas de dados homogêneas

Muitas vezes precisamos de armazenar uma quantidade arbitrária de números para resolver um problema. Por exemplo, para calcular a média de vários números podemos ler um valor por vez e armazenar a soma destes valores, depois retornar a soma dividida pela quantidade de valores. Porém, se quisermos o desvio padrão desses valores, precisaríamos calcular a média dos valores e depois revisitar cada valor lido. É inviável ler os valores duas vezes. Uma solução melhor é armazenar temporariamente os valores lidos para fazer todos os cálculos necessários para retornar a resposta desejada. Vetores são a forma mais simples de armazenar uma quantidade arbitrária de valores em linguagem C.

4.1 Vetores

A quantidade de situações em que precisamos agrupar vários elementos de mesmo tipo é vasta. Podemos armazenar estes elementos em estruturas de dados homogêneas como vetores. O programa 53 mostra uma forma de criar um vetor com quatro valores.

Programa 53: Declaração de um vetor com quatro valores predefinidos.

```
1  #include <stdio.h>
2  int main() {
3    float vet[] = {2, 0, 1, 9};
4    int i;
5    for (i=0; i<4; i++) {
6      printf("%.2f ", vet[i]);
7    }
8    printf("\n");
9    return 0;
10 }
```

Um vetor é composto por posições de memória contíguas do mesmo tipo. A linha 3 do programa 53 declara um vetor de `float` com os valores 2, 0, 1 e 9. Nessa forma de declarar um vetor em C, usa-se os colchetes "[ ]" vazios e é obrigatório que os valores do vetor estejam separados por vírgula e entre chaves "{}". Para acessar os valores de um vetor, usa-se o nome dado ao vetor em sua declaração com um índice inteiro entre os colchetes. Os índices válidos vão de 0 (zero) até $n - 1$, onde n é a quantidade de elementos do vetor. Por exemplo, `vet[0]` acessa o primeiro elemento do vetor `vet` do programa 53, e `vet[3]` acessa o último elemento. Pode-se usar variáveis inteiras para acessar os elementos de um vetor, como mostra a linha 6 do programa 53. A atribuição de valores a um vetor segue a mesma lógica do acesso a seus elementos. Por exemplo, o código `vet[1]=5;` atribui o valor 5 à segunda posição do vetor.

A declaração de vetores apresentada no programa 53 é útil para testes pequenos ou outras raras situações onde o vetor armazena uma quantidade pequena de valores já conhecidos. Nas situações mais comuns de programação, não conhecemos os valores do vetor de antemão ou a quantidade de valores é tão grande que fica inviável declarar um vetor com valores aleatoriamente digitados. Nestas situações, é melhor declarar os vetores como mostra o exemplo do programa 54.

Programa 54: Declaração de um vetor com quatro valores.

```
1  #include <stdio.h>
2  int main() {
3    float vet[4];
4    int i;
5    for (i=0; i<4; i++) {
6      vet[i] = 0;
7    }
8    for (i=0; i<4; i++) {
9      printf("%.2f ", vet[i]);
10   }
11   printf("\n");
12   return 0;
13 }
```

Na declaração do vetor como mostra a linha 3 do programa 54 é necessário que tenha um valor inteiro entre os colchetes, representando a quantidade de valores que o vetor armazenará. Neste caso, os valores contidos no vetor são indefinidos. É possível que todos os valores sejam zero, em alguma execução, mas não existe garantia de que sempre serão zero. Por isso, caso você queira que os valores do vetor comecem com zero, você deve fazer isso usando um laço de repetição, como mostram as linhas 5, 6 e 7. O programa 55 mostra mais uma forma de se declarar um vetor em C.

Programa 55: Declaração e leitura de um vetor com quatro valores.

```
1  #include <stdio.h>
2  int main() {
3    const int tam = 4;
4    float vet[tam];
5    int i;
6    for (i=0; i<tam; i++) {
7      scanf("%f", &vet[i]);
8    }
9    for (i=0; i<tam; i++) {
10     printf("%.2f ", vet[i]);
11   }
12   printf("\n");
13   return 0;
14 }
```

As versões mais modernas da linguagem C e seus compiladores aceitam variáveis entre os colchetes em uma declaração de vetor. Porém, isso pode não funcionar em compiladores mais antigos. Por isso, a palavra `const` aparece na linha 3 do programa 55 para indicar que `tam` é uma constante. Se quisermos que o programa rode para mais valores do que 4 basta alterarmos a linha 3. A linha 7 do programa 55 mostra como fazer a leitura pelo teclado de elementos de um vetor de `float` usando `scanf`. Entretanto, quando estamos testando um programa, é bem improdutivo digitar os valores a cada teste. O console 18 mostra como entrar com os valores mais rapidamente por um console.

Console 18: Compilando e executando o programa 55 com entrada de valores.

```
1 $ gcc vetor.c -o vetor.bin
2 $ $./vetor.bin <<< "4 5 3 2"
3 4.00 5.00 3.00 2.00
4 $
```

4.1.1 Vetores passados por parâmetro

Uma função para retornar a média da turma poderia ser:

Algoritmo 1: media_vet_float

```
1  float media_vet_float(float vet[], int tam) {
2    int i;
3    float soma = 0.0f;
4    for(i=0; i<tam; i++) {
5      soma += vet[i];
6    }
7    return soma/tam;
8  }
```

A assinatura da função também poderia ser:

```
float media_vet_float(float *vet, int tam);
```

O código para imprimir a média da turma na tela:

```
printf("A media da turma e %f\n", media_vet_float(notas, quant_alunos));
```

4.1.2 Alocação dinâmica de vetores

Seguindo a ideia do do exemplo anterior, teríamos um problema se não soubéssemos a quantidade de alunos da turma no momento da implementação do programa. Este problema pode ser facilmente resolvido se soubermos a quantidade de alunos durante a execução do programa. Para isso, devemos utilizar a alocação dinâmica através das funções `malloc` e `free`.

Algoritmo 2: Vetores dinâmicos

```
1  int main() {
2    int quant_alunos;
3    float* notas;
4    printf("Quantidade de alunos: ");
5    scanf("%d", &quant_alunos);
6    notas = malloc(sizeof(float)*quant_alunos);
7    printf("Preenchimento da nota:\n");
8    preenche_vet_float(notas, quant_alunos);
9    printf("Impressao da nota:\n");
10   print_vet_float(notas, quant_alunos);
11   printf("A media da turma e %f\n", media_vet_float(notas, quant_alunos));
12   free(notas);
13   return 0;
14 }
```

As funções *preenche_vet_float* e *print_vet_float* devem ser implementadas como exercício.

A situação ficaria um pouco mais complicada se cada aluno tivesse mais de uma nota na disciplina. Por exemplo, se cada aluno tiver duas notas na disciplina, teremos que usar dois vetores para armazenar as notas. Seguindo este pensamento, teríamos que alocar uma quantidade de vetores igual a quantidade de notas de cada aluno.

Fica como exercício implementar um programa com duas notas para cada aluno utilizando dois vetores.

Uma solução engenhosa poderia nos permitir alocar apenas um vetor duas vezes maior que a quantidade de alunos para armazenar as duas notas de cada aluno, mas esta implementação não seria muito legível.

Algoritmo 3: Um vetor para duas notas

```
1  int main() {
2    int quant_alunos;
3    float* notas;
4    printf("Quantidade de alunos: ");
5    scanf("%d", &quant_alunos);
6    notas = malloc(sizeof(float)*quant_alunos*2);
7    printf("Preenchimento da primeira nota:\n");
8    preenche_vet_float(notas, quant_alunos);
9    printf("Preenchimento da segunda nota:\n");
10   preenche_vet_float(notas+quant_alunos, quant_alunos);
11   printf("Impressao da primeira nota:\n");
12   print_vet_float(notas, quant_alunos);
13   printf("Impressao da segunda nota:\n");
14   print_vet_float(notas+quant_alunos, quant_alunos);
15   printf("A media da turma e %f\n", media_vet_float(notas, quant_alunos));
16   free(notas);
17   return 0;
18 }
```

4.1.3 Exercícios

Para cada questão abaixo, faça um programa para testá-la.

1. Implemente um procedimento que receba como parâmetro um vetor de números reais (vet) de tamanho n e imprima estes valores na tela.

2. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n e retorne quantos números negativos estão armazenados nesse vetor. Esta função deve obedecer ao protótipo:
   `int negativos (int n, float* vet);`

3. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n e retorne quantos números pares estão armazenados nesse vetor. Esta função deve obedecer ao protótipo:
   `int pares (int n, float* vet);`

4. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n e retorne a soma dos números armazenados nesse vetor. Esta função deve obedecer ao protótipo:
   `float soma (int n, float* vet);`

5. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n e retorne a média dos números armazenados nesse vetor. Esta função deve obedecer ao protótipo:
   `float media (int n, float* vet);`

6. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n, um número real x e multiplique números armazenados nesse vetor por x. Esta função deve obedecer ao protótipo:
   `void multiplica_por_escalar(int n,float* vet,float x);`

7. Implemente uma função que permita a avaliação de polinômios. Cada polinômio é definido por um vetor que contém seus coeficientes. Por exemplo, o polinômio de grau 2, 3x2 + 2x + 12, terá um vetor de coeficientes igual a v[ ] = 12, 2, 3. A função deve obedecer ao protótipo:
   `double avalia (double* poli, int grau, double x);`
   Onde o parâmetro poli é o vetor com os coeficientes, grau é o grau do polinômio e x é o valor para o qual o polinômio deve ser avaliado.

8. Implemente uma função que receba 3 vetores de números reais de mesmo tamanho e coloque a soma dos dois primeiros no terceiro.

9. Implemente uma função que receba 2 vetores de números reais de mesmo tamanho e coloque no segundo a diferença entre a média e cada elemento do primeiro.

10. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n e retorne desvio padrão dos números armazenados nesse vetor. Esta função deve obedecer ao protótipo: `float desv_pad (int n, float* vet);`

11. Implemente uma função que receba 2 vetores de números reais de mesmo tamanho, um número real x e some ao segundo vetor os elementos do primeiro multiplicados por x.

12. Implemente uma função que receba 2 vetores de números reais de mesmo tamanho e some ao primeiro elemento do segundo vetor o último elemento do primeiro, ao segundo elemento do segundo vetor o penúltimo elemento do primeiro e assim sucessivamente.

13. Implemente uma função que receba um vetor de n notas e a nota mínima para aprovação e retorne a porcentagem de alunos que foram aprovados.

14. Implemente uma função recursiva que inverta a ordem dos elementos de um vetor de números reais.

15. Implemente uma função que coloque em ordem não decrescente um vetor de números reais de tamanho n.

4.2 Matrizes

Vamos agora voltar ao exemplo com alunos e avaliações, só que agora, cada aluno terá 3 (três) notas e teremos somente 4 alunos para diminuir a digitação necessária. Felizmente a linguagem C não nos obriga a utilizar três vetores de 4 posições e nem criar códigos tão confusos quanto o último. É verdade que C não é a mais legível das linguagens de programação, mas a legibilidade do código depende mais do estilo de programação do que da linguagem em si. Para este exemplo poderíamos utilizar uma matriz 4x3. Desta forma, podemos armazenar, de forma legível, as três notas de cada aluno em uma única estrutura, como mostra o algoritmo 4.

Algoritmo 4: Matrizes estáticas

```
1  #include <stdio.h">
2  int main(){
3    const int quant_alunos=4, quant_notas=3;
4    int i, j;
5    float notas[quant_alunos][quant_notas];
6    printf("Preenchimento das notas:\n");
7    for(i=0; i<quant_alunos; i++){
8      printf("\tAluno %d: \n", i+1);
9      for(j=0; j<quant_notas; j++){
10       printf("Nota %d: ", j+1);
11       scanf("%f", &notas[i][j]);
12     }
13   }
14   printf("Impressao das notas:\n");
15   for(i=0; i<quant_alunos; i++){
16     printf("Aluno %d: ", i+1);
17     for(j=0; j<quant_notas; j++){
18       printf("%5.1f ", notas[i][j]);
19     }
20     printf("\n");
21   }
22   return 0;
23 }
```

Esta representação nos permite armazenar valores em linhas e colunas de forma clara. O código da quarta linha do programa acima (*float notas[quant_alunos][quant_notas];*) [1] declara uma matriz de 4 linhas e 3 colunas. Desta forma, temos uma linha para cada aluno e uma coluna para cada nota.

O código para atribuir 10 a primeira nota do primeiro aluno é:

```
notas[0][0] = 10;
```

De forma análoga, o código para imprimir a última nota do último aluno é:

```
printf("%f", notas[3][2]);
```

4.2.1 Passagem de matrizes como argumento

Se visarmos uma melhor organização de nossos programas e reuso de código, inevitavelmente teremos que utilizar funções que recebem matrizes como parâmetro. Exemplos disso são as funções do algoritmo 5 (`preenche_mat_float`, `imprime_mat_float` e `media_mat_float`). Este algoritmo é muito mais legível do que o algoritmo 4, porém a passagem de matrizes como parâmetro não é tão trivial como a passagem de vetores.

Algoritmo 5: Funções com matrizes estáticas

```
1  #include <stdio.h>
2  int main() {
3    const int quant_alunos=32, quant_notas=3;
4    float notas[quant_alunos][quant_notas];
5    printf("Preenchimento das notas:\n");
6    preenche_mat_float(notas, quant_alunos, quant_notas);
7    printf("Impressao das notas:\n");
8    imprime_mat_float(notas, quant_alunos, quant_notas);
9    printf("A media da turma e %f\n", media_mat_float(notas, quant_alunos,
       quant_notas));
10   return 0;
11 }
```

Já conhecemos duas formas de se passar vetores como parâmetro. As duas formas aceitam tanto vetores alocados de forma estática, no começo do programa, quanto vetores alocados de forma dinâmica, utilizando a função `malloc`. Apesar de todas as vantagens que as matrizes trazem em certas situações, elas não facilitam a vida do programador quando este precisa passá-las como parâmetro para uma função. Quando uma matriz é alocada de forma estática, a função que irá recebê-la como parâmetro deve saber de antemão quantas colunas a matriz terá. Isto pode ser muito inconveniente, uma função feita para uma matriz de 3 colunas só poderá receber matrizes com esta quantidade de colunas. Por exemplo, função `media_mat_float` deve ser implementada como mostra o algoritmo 6.

Algoritmo 6: Calcula média da matriz

```
1  float media_mat_float(float mat[][3], int lin, int col) {
2    int i, j;
3    float soma = 0;
4    for(i=0; i<lin; i++) {
5      for(j=0; j<col; j++) {
6        soma += mat[i][j];
7      }
8    }
9    return soma/(lin*col);
10 }
```

[1] Por padrão, a declaração de matrizes e vetores estáticos não permite variáveis dentro dos colchetes. Por isso a palavra `const` antes da declaração de `quant_alunos` e `quant_notas`.

Isto é decepcionante, pois é a única forma de se passar uma matriz aloca de forma estática como parâmetro. Além disso, esta função ficou tão restrita que não podemos usá-la se resolvermos usar duas ou quatro notas ao invés de três. Para fazermos funções com utilizações mais amplas precisamos alocar as matrizes de forma dinâmica.

4.2.2 Alocação dinâmica de matrizes

De uma forma análoga a um vetor, uma matriz pode ser imaginada com um vetor de vetores. Por exemplo uma matriz float 32x3 pode ser vista como um vetor de 32 posições do tipo vetor de float com 3 posições. De outra forma, o vetor float mat[32][3] possui 32 vetores float[3], que por sua vez, possui 3 float.

Lembrando que um vetor pode ser alocado, de forma dinâmica, da seguinte forma:

```
float* notas;
notas = malloc(sizeof(float)*32);
```

Um vetor de vetores pode ser alocado da seguinte forma:

```
float** notas;
notas = malloc(sizeof(float*)*32);
```

Isto aloca um vetor com 32 posições de ponteiros para float. Como sabemos, um vetor é referenciado por um ponteiro para sua primeira posição. Então, também devemos alocar um vetor de 3 posições de float para cada posição do vetor de vetores notas:

```
int i;
float** notas;
notas = malloc(sizeof(float*)*32);
for(i=0; i<32; i++) {
  notas[i] = malloc(sizeof(float)*3);
}
```

Assim o programa principal seria alterado como mostra o algoritmo 7. Cada posição de memória alocada com a função `malloc` precisa ser liberada pela função `free`. isso acontece nas linhas 16 a 19 do algoritmo. Detalhes da alocação e liberação de matrizes dinâmicas são explicados com os algoritmos 9 e 10.

Algoritmo 7: Funções com matrizes estáticas

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4    const int quant_alunos=32, quant_notas=3;
5    int i;
6    float** notas;
7    notas = malloc(sizeof(float*)*quant_alunos);
8    for(i=0; i<quant_alunos; i++) {
9      notas[i] = malloc(sizeof(float)*quant_notas);
10   }
11   printf("Preenchimento das notas:\n");
12   preenche_mat_float(notas, quant_alunos, quant_notas);
13   printf("Impressao das notas:\n");
14   imprime_mat_float(notas, quant_alunos, quant_notas);
15   printf("A media da turma e %f\n", media_mat_float(notas, quant_alunos,
       quant_notas));
16   for(i=0; i<quant_alunos; i++) {
17     free(notas[i]);
18   }
19   free(notas);
20   return 0;
21 }
```

Agora a função `media_mat_float` pode ser implementada como mostra o algoritmo 8. Nota-se que a função ficou muito mais genérica, e agora pode calcular a média para qualquer quantidade de alunos e notas ou alguma matriz de float que não represente alunos e notas.

Algoritmo 8: Calcula média de matriz dinâmica.

```
1  float media_mat_float(float** mat, int lin, int col) {
2    int i, j;
3    float soma = 0;
4    for(i=0; i<lin; i++) {
5      for(j=0; j<col; j++) {
6        soma += mat[i][j];
7      }
8    }
9    return soma/(lin*col);
10 }
```

Alocar matrizes de forma dinâmica pode ser cansativo quando for muito frequente. Nestes casos, é interessante criar uma função para alocar uma matriz, como mostra o algoritmo 9.

Algoritmo 9: Aloca matriz dinâmica.

```
1  float** aloca_mat_float(int lin, int col) {
2    int i;
3    float** mat;
4    mat = malloc(sizeof(float*)*lin);
5    for(i=0; i<lin; i++) {
6      mat[i] = malloc(sizeof(float)*col);
7    }
8    return mat;
9  }
```

Não podemos liberar o espaço de memória ocupado pela matriz simplesmente usando a função free diretamente na matriz, neste caso, notas. Se assim fizermos, liberaremos apenas o vetor de vetores, e não os vetores propriamente ditos. O espaço de memória dos 32 vetores de 3 float continuará sendo ocupado por eles. Temos que primeiro liberar cada vetor de float da matriz para depois liberá-la. Como isso pode ser tão trabalhoso quanto alocar uma matriz, é interessante que façamos uma função para liberar a matriz:

Algoritmo 10: Libera matriz dinâmica

```
1  void libera_mat_float(float** mat, int lin) {
2    int i;
3    for(i=0; i<lin; i++) {
4      free(mat[i]);
5    }
6    free(mat);
7  }
```

Se não soubermos nem a quantidade de alunos e nem a quantidade de notas de cada aluno durante a implementação de um programa, podemos implementá-lo como no algoritmo 11.

4.2.3 Exercícios

Para cada função abaixo faça um programa principal para testá-la.

1. Implemente uma função que aloque uma matriz de `float`.
   `float** aloca_matriz(int nlinhas, int ncolunas);`

Algoritmo 11: Calcula média da turma

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4    int quant_alunos, quant_notas;
5    float** notas;
6    printf("Quantidade de alunos: ");
7    scanf("%d", &quant_alunos);
8    printf("Quantidade de notas: ");
9    scanf("%d", &quant_notas);
10   notas = aloca_mat_float(quant_alunos, quant_notas);
11   printf("Preenchimento das notas:\n");
12   preenche_mat_float(notas, quant_alunos, quant_notas);
13   printf("Impressao das notas:\n");
14   imprime_mat_float(notas, quant_alunos, quant_notas);
15   printf("A media da turma e %f\n", media_mat_float(notas, quant_alunos,
         quant_notas));
16   libera_mat_float(notas, quant_alunos);
17   return 0;
18 }
```

2. Implemente um procedimento que preencha uma matriz de `float` com valores aleatórios.
   `void preeche_matriz(int nlinhas, int ncolunas, float** matriz);`

3. Implemente um procedimento que imprima uma matriz de `float`.
   `void imprime_matriz(int nlinhas, int ncolunas, float** matriz);`

4. Implemente um procedimento que libere uma matriz de `float` alocada dinamicamente.
   `void libera_matriz(int nlinhas, float** matriz);`

5. Implemente uma função que retorne a soma dos elementos de uma matriz de `float`.
   `float soma_elementos_matriz(int nlinhas, int ncolunas, float** matriz);`

6. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}6 & 5 & 4\\3 & 2 & 1\end{bmatrix}$$

7. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}3 & 2 & 1\\6 & 5 & 4\end{bmatrix}$$

8. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}4 & 5 & 6\\1 & 2 & 3\end{bmatrix}$$

9. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 4\\2 & 5\\3 & 6\end{bmatrix}$$

10. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}4 & 1\\5 & 2\\6 & 3\end{bmatrix}$$

11. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}6 & 3\\5 & 2\\4 & 1\end{bmatrix}$$

12. Implemente um procedimento que imprima uma matriz de `float` da seguinte forma:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \Rightarrow \begin{bmatrix}3 & 6\\2 & 5\\1 & 4\end{bmatrix}$$

13. Implemente uma função que retorne a soma dos elementos da diagonal principal de uma matriz quadrada.

14. Implemente uma função que retorne a soma dos elementos da diagonal secundária de uma matriz quadrada.

15. Implemente um procedimento que recebe uma matriz $m \times n$, um vetor de tamanho m, e coloque a média de cada linha nas respectivas posições do vetor.

16. Implemente um procedimento que recebe uma matriz $m \times n$, um vetor de tamanho n, e coloque a média de cada coluna nas respectivas posições do vetor.

17. Implemente um procedimento que receba uma matriz e dois índices representando linhas dessa matriz. Esse procedimento deve somar as linhas dos respectivos índices e guardar o resultado na linha do primeiro índice. O exemplo seguinte mostra a soma das linhas 2 e 1 da matriz.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2 & 3\\5 & 7 & 9\\7 & 8 & 9\end{bmatrix}$$

18. Análogo ao exercício anterior, implemente um procedimentos que some duas colunas de uma matriz. O exemplo seguinte soma as colunas 1 e 2 da matriz.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix} \Rightarrow \begin{bmatrix}3 & 2 & 3\\9 & 5 & 6\\15 & 8 & 9\end{bmatrix}$$

19. Faça uma função que some duas matrizes de mesma ordem.
```
float** soma_matrizes(float** matA, float** matB, int linhas, int colunas);
```

20. Faça uma função que multiplique uma matriz por um escalar.
```
void multiplica_matriz_escalar(float** mat, int linhas, int colunas, float escalar);
```

21. Faça uma função que multiplique duas matrizes, se possível.
```
float** multiplica_matrizes(float** matA, int linhasA, int colunasA,
float** matB, int linhasB, int colunasB);
```

22. Faça uma função que retorne a transposta de um matriz.
```
float** transposta(float** mat,int linhas,int colunas);
```

23. Faça uma função que retorne a matriz de menor complemento de um elemento de uma matriz quadrada.
```
float** menor_complemento(float** mat, int ordem, int linhaElemento, int colunaElemento);
```

24. Faça uma função que retorne o determinante de uma matriz quadrada.
```
float determinante(float** mat, int ordem);
```

25. Faça uma função que retorne a matriz de cofatores de uma matriz quadrada.
```
float** cofatores(float** mat, int ordem);
```

26. Faça uma função que retorne a matriz adjunta de uma matriz quadrada.
```
float** adjunta(float** mat, int ordem);
```

27. Faça uma função que retorne a matriz inversa de uma matriz quadrada, se possível.
    `float** inversa(float** mat, int ordem);`

28. Faça uma função que retorne a pseudo-inversa de uma matriz qualquer.
    `float** pseudoinversa(float** mat, int linhas, int colunas);`

4.3 Strings

Strings A linguagem C não um tipo cadeia de caracteres (string). Na verdade, a linguagem C não possui sequer um tipo caractere. A representação de um caractere é feita por um inteiro de 8 bits (char). Por isso, para representamos cadeias de caracteres em C utilizamos vetores do tipo char, com o caractere '\0' marcando o fim dos caracteres válidos. Por exemplo, para representar a string "Hello world", podemos utilizar um vetor de 20 posições de char, mas ocuparemos apenas as primeiras 11 posições. 10 posições para o texto válido e 1 posição para marcar o fim do texto válido com o caractere '\0'. Os caracteres contidos no vetor após o caractere '\0' não interessam para esta representação de string.

O algoritmo 12 mostra três formas de se atribuir uma string a um vetor de caractere.

Algoritmo 12: Exemplo com strings.

```
1  #include <stdio.h>
2  int main() {
3    char str1[10];
4    str1[0] = 'H';
5    str1[1] = 'a';
6    str1[2] = 'r';
7    str1[3] = 'd';
8    str1[4] = '\0';
9    char str2[] = {'H','e','l','l','o','\0'};
10   char str3[] = "World";
11   printf("%s %s %s", str1, str2, str3);
12 }
```

Fica claro que a terceira forma de atribuição de uma string a um vetor é a opção mais simples. Porém, a cópia de strings de um vetor para outro não é tão trivial, precisando de funções para facilitar a programação com strings.

Funções

A biblioteca `string.h` possui várias funções para fazer operações com este tipo de estrutura. Algumas das mais usadas são:

strcat

Declaração/Assinatura:
`char *strcat(char *str1, const char *str2);`

Anexa a string apontada por str2 no final da string apontada por str1. A terminação com o caractere null ('\0') de str1 é sobreescrita. A cópia para quando o caractere null de str2 é copiado. Se uma sobreposição ocorrer, o resultado é indefinido. O argumento str1 é retornado.

Exemplo:

```
#include <stdio.h>
#include <string.h>
int main() {
  char str1[] = "Hello";
  char str2[] = "World";
  strcat(str1,str2);
  printf(str1);
  return 0;
}
```

Resultado: `HelloWorld`.

strcmp

Declaração:

```
int strcmp(const char *str1, const char *str2);
```

Compara a string apontada por str1 com a string apontada por str2. Retorna zero se str1 e str2 são iguais. Retorna menor que zero ou maior que zero se str1 é menor ou maior que str2, respectivamente.

Exemplo:

```
#include <stdio.h>
#include <string.h>
int main() {
  char str1[] = "Hello";
  char str2[] = "World";
  printf("%d",strcmp(str1,str2));
  return 0;
}
```

Resultado: `-15`.

strcpy

Declaração:

```
char *strcpy(char *str1, const char *str2);
```

Copia a string apontada por str2 para str1. Copia até o caractere null, inclusive. Se str1 e str2 forem sobrepostas o comportamento é indefinido. Retorna o argumento str1.

Exemplo:

```
#include <stdio.h>
#include <string.h>
int main() {
  char str1[] = "Hello";
  char str2[] = "World";
  strcpy(str1,str2);
  printf(str1);
  return 0;
}
```

Resultado: `World`.

strlen

Declaração:

```
size_t strlen(const char *str);
```

Computa o comprimento da string str até o caractere de terminação null, mas não o inclui. Retorna o número de caracteres da string.

Exemplo:

```
#include <stdio.h>
#include <string.h>
int main() {
  char str[] = "Hello World";
  printf("%d",strlen(str));
  return 0;
}
```

Resultado: `11`.

Para ver todas as funções da biblioteca `string.h` acesse http://www.cplusplus.com/reference/cstring/.

Exercícios

1. Faça um procedimento que imprima em decimal (%d) as 256 possibilidades de valor para uma variável do tipo char.
2. Faça um procedimento que imprima em decimal (%d) as 256 possibilidades de valor para uma variável do tipo unsigned char.
3. Faça um procedimento que imprima em hexadecimal (%x) as 256 possibilidades de valor para uma variável do tipo unsigned char.
4. Faça um procedimento que imprima em octal (%o) as 256 possibilidades de valor para uma variável do tipo unsigned char.
5. Faça um procedimento que imprima a tabela ASCII formatada com 10 colunas.
6. Faça um procedimento que imprima uma string passada como parâmetro, sem utilizar %s.
7. Faça uma função que retorne o tamanho de um string, sem usar a biblitoca `string.h`.
8. Faça um procedimento que receba duas strings como parâmetros e copie o valor da 2ª para a 1ª, sem usar a biblitoca `string.h`.
9. Faça um procedimento que receba duas strings como parâmetros e concatene a 2ª à 1ª, sem usar a biblitoca `string.h`.
10. Faça uma função que retorne a quantidade de vogais de uma string.
11. Implemente uma função que receba como parâmetros uma string e um caractere e retorne o número de ocorrências desse caractere.
12. Implemente uma função que receba uma string como parâmetro e altere nela as ocorrências de caracteres maiúsculos para minúsculos.
13. Implemente uma função que receba uma string como parâmetro e substitua todas as letras por suas sucessoras no alfabeto. Por exemplo, a string "Casa"sera alterada para "Dbtb". A letra z deve ser substituída por a (e Z por A). Caracteres que não forem letras devem permanecer inalterados.
14. Implemente uma função que receba uma string como parâmetro e substitua as ocorrências de uma letra pelo seu oposto no alfabeto, isto é, a↔z, b↔y, c↔x etc. Caracteres que não forem letras devem permanecer inalterados.
15. Implemente uma função que receba uma string como parâmetro e desloque os seus caracteres uma posição para direita. Por exemplo, a string "casa"será alterada para "acas". Note que o último caractere vai para o início da string.
16. Faça uma função que retorne uma cópia de uma string. Reimplemente as funções dos exercícios 12 a 15 para que retornem uma nova string, alocada dentro da função, com o resultado esperado, preservando as strings originais inalteradas. Essas funções devem obedecer ao seguinte protótipo:
    ```
    char* nome_da_funcao(char* str);
    ```
17. Para cada função e procedimento dos exercícios sobre string anteriores, faça um programa principal que os teste, fazendo a leitura da string a ser testada pelo teclado. O código para ler uma string de, por exemplo, 30 caracteres pode ser:
    ```
    char string[31];
    scanf(" %30[^\n]", string);
    ```

4.4 Conjunto de strings

Não seria difícil encontrar situações em que seria necessário armazenar um conjunto de strings. Se considerarmos que string é uma estrutura singular, o armazenamento de um conjunto de strings pode ser feito em um vetor. Sabemos uma string é guardada vetor do tipo char. Então, para termos um vetor de string, teremos na verdade um vetor de vetores do tipo char, ou um matriz do tipo char.

Algoritmo 13: Lê 5 nomes e os imprime em ordem alfabética

```
1 int main() {
2   int vet_tam = 5;
3   char vet_string[5][31];
4   preenche_vet_string(vet_string, vet_tam);
5   ordena_vet_string(vet_string, vet_tam);
6   print_vet_string(vet_string, vet_tam);
7   return 0;
8 }
```

Vamos exemplificar o uso de uma matriz do tipo char com um programa que lê 5 nomes de até 30 caracteres e os imprime em ordem alfabética.

Note que consideramos apenas 1 tamanho, que é a quantidade de linhas, por que determinamos que as strings teriam no máximo 30 caracteres. Lembre-se que para armazenar uma string de 30 caracteres precisamos de um vetor de pelo menos 31 caracteres, por causa do '\0'.

Para entender como seria a passagem por parâmetro dessa lista de nomes, vamos ver uma implementação da função print_vet_string.

Algoritmo 14: Imprime um vetor de strings

```
1 void print_vet_string(char vet_string[][31], int vet_tam) {
2   int i;
3   for(i=0; i<vet_tam; i++) {
4     printf("%s\n", vet_string[i]);
5   }
6 }
```

Veja que definimos o tamanho máximo das strings na declaração da função. Isto não é um grande problema, já que as strings não precisam ter 30 caracteres. Este é seu tamanho máximo.

As funções para preencher e ordenar o vetor de strings devem ser implementadas como exercício.

Um problema mais complexo aparece se não soubermos a quantidade de strings que teremos que armazenar durante a implementação do programa. Neste caso teríamos que alocar o vetor de strings dinamicamente, o que não permitiria a passagem de vet_string no formato vet_string[ ][31], nos obrigando a usar uma notação de vetor de vetores do tipo **vet_string.

Alocação Dinâmica

Sabemos que alocação de matrizes de forma dinâmica pode ser cansativa. Com vetores de strings não seria diferente. Então, é interessante que implementemos funções para alocar e liberar vetores de strings. A função de alocação pode ser implementada da seguinte forma:

Algoritmo 15: Aloca um vetor de strings

```
1 char** aloca_vet_string(int vet_tam, int str_tam) {
2   int i;
3   char** vet_string;
4   vet_string = malloc(sizeof(char*)*vet_tam);
5   for(i=0; i<vet_tam; i++) {
6     vet_string[i] = malloc(sizeof(char)*str_tam);
7   }
8   return vet_string;
9 }
```

A função para liberação de vetor de strings é análoga a função de liberação de matrizes, portanto, deve

ser implementada como exercício. Neste caso o programa principal poderia ser implementado da seguinte forma:

Algoritmo 16: Imprime um vetor de strings

```
1  int main() {
2    int vet_tam, str_tam;
3    char** vet_string;
4    string_tam = 31;
5    printf("Quantidade de nomes: ");
6    scanf("%d", &vet_tam);
7    vet_string = aloca_vet_string(vet_tam, str_tam);
8    preenche_vet_string(vet_string, vet_tam);
9    ordena_vet_string(vet_string, vet_tam);
10   print_vet_string(vet_string, vet_tam);
11   libera_vet_string(vet_string, vet_tam);
12   vet_string = NULL;
13   return 0;
14 }
```

As funções para preencher, ordenar e imprimir devem ser alteradas para receber o vetor de strings no formato **vet_string. Veja como ficaria a função print_vet_string:

Algoritmo 17: Imprime um vetor de strings

```
1 void print_vet_string(char** vet_string, int vet_tam) {
2   int i;
3   for(i=0; i<vet_tam; i++) {
4     printf("%s\n", vet_string[i]);
5   }
6 }
```

Para exercitar, implemente as funções que não foram escritas aqui e teste o programa.

Exercícios

Exercícios com matrizes e vetores de string.

1. Uma dona de casa gostaria de ter um programa em que ela pudesse inserir sua lista de compras e as cotações de cada produto em cada supermercado pesquisado. Após a inserção dos dados, ela gostaria que o programa informasse qual supermercado a compra total custaria menos e qual valor que ela pagaria.

2. Atualize o programa anterior para que ele informe não apenas o supermercado mais barato, mas uma lista com os supermercados e respectivos custos de compra ordenada pelo custo da compra.

3. Faça mais uma atualização no programa, para que ele informe o preço mínimo, máximo e médio de cada produto.

4. Faça um programa que receba uma lista com nomes de alunos, as notas de cada aluno e a nota mínima para aprovação na disciplina. O aluno é considerado aprovado se a média de suas notas forma maior ou igual a nota mínima para aprovação. O programa deve informar uma lista de alunos aprovados e outra de alunos reprovados.

5. Em um concurso de escolas de samba são avaliados vários quesitos por vários juízes. A nota em cada quesito é a soma das notas dos juízes, descartando-se a maior e a menor nota. A vencedora é aquela que conseguir a maior nota total, que é a soma acumulada em todos os quesitos. Faça um programa que receba uma lista de escolas de samba, uma listas de quesitos e a nota de cada juiz em seu respectivo

quesito. O programa deve apresentar uma tabela com a nota de cada quesito para cada escola de samba. A tabela deve estar ordenada pela classificação geral de cada escola de samba.

4.5 Parâmetros da função principal

A linguagem C pode aceitar parâmetros, como mostra o exemplo abaixo:

Algoritmo 18: Imprime parâmetros da função principal

```
1 int main(int argc, char** argv){
2   int i;
3   printf("%d\n", argc);
4   for(i=0; i<argc; i++){
5     printf("%s\n", argv[i]);
6   }
7   return 0;
8 }
```

Os parâmetros devem ser um int e um char**, necessariamente nesta ordem. Os nomes argc e argv não são obrigatórios, mas são um padrão de programação. O parâmetro argc guarda a quantidade de argumentos passados para a chamada do programa principal. O parâmetro argv é um vetor de strings que guarda os valores de cada string passada como argumento para o programa principal. Se compilarmos o código acima gerando um programa de nome teste e executarmos este programa, ele dará a seguinte resposta:

```
$ ./teste
1
./teste
```

Um programa em C entende que o nome do programa chamado é um argumento. Por isso, o código acima gera um programa que exibe o valor 1 para argc e imprime a string na posição 0 do vetor de strings argv. Portanto, a primeira posição de argv sempre terá o nome do programa. Se chamarmos o programa de fora do seu diretório o resultado será os seguinte: `$ ./Aula/teste`

```
1
./Aula/teste
```

Note que o programa considera também o caminho para o executável como parte do nome do programa. Podemos passar qualquer quantidade de argumentos para o programa, mas todos serão considerados strings, independente serem letras ou números. Os argumentos são separados por espaço.

```
$ ./teste 54 10,6 32.1 programa de computador
7
./teste
54
10,6
32.1
programa
de
Computador
```

Note que o uso de ponto ou de vírgula não importa, pois o programa considera que tudo é string. Se quisermos um argumento com espaços, como por exemplo, "programa de computador", devemos usar aspas, simples ou duplas, para o argumento. Mas se você utilizar aspas simples na abertura do argumento, ele deverá ser fechado com aspas simples. O mesmo acontece com as aspas duplas.

```
$ ./teste 54 10,6 32.1 "programa de computador"
5
./teste
54
10,6
32.1
programa de computador
```

Utilizando esta funcionalidade, podemos passar alguns valores necessários para a execução do programa sem a utilização de scanf. Não é possível passar argumento do tipo int para um programa pela sua chamada,

pois o parâmetro argv só aceita strings. Mas podemos transformar esta string em um inteiro com a função atoi da biblioteca stdlib.h. Veja o código levemente alterado para a nova funcionalidade:

Algoritmo 19: Imprime argumentos inteiros

```
1 int main(int argc, char** argv){
2   int i;
3   printf("%d\n", argc);
4   for(i=0; i<argc; i++)
5   {
6     printf("%d\n", atoi(argv[i]));
7   }
8   return 0;
9 }
```

Veja agora um exemplo da execução do programa modificado:

```
$ ./teste 15 64 a
4
0
15
64
0
```

O número 4 representa a quantidade de argumentos. O zero que vem logo depois é o retorno da função atoi tendo como parâmetro a string "./teste". Os números 15 e 64 foram convertidos corretamente. Já o caractere a retornou zero. Caso você queira um argumento do tipo float, você pode usar a função atof, também da biblioteca stdlib.h.

Para exercitar o conteúdo apresentado aqui, faça, usando argumentos de entrada via linha de comando, programas simples como: Conversão de fahrenheit para celsius e vice-versa; Conversão de milímetros para polegada, quilômetros para milhas etc; Área do triângulo, retângulo e quadrado; E mais quantos programas você imaginar.

Capítulo 5

Estruturas de dados Heterogêneas

Quando estamos estudando programação, a maioria dos problemas propostos é muito simples, tal que possamos solucioná-los com pouco conhecimento da linguagem que estamos aprendendo. Problemas mais complexos podem ser resolvidos utilizando apenas estes conhecimentos básicos, porém, a solução pode ser confusa, além de ser, certamente, muito acoplada e pouco coesa. Soluções de problemas complexos que utilizam apenas os tipos fornecidos pela linguagem, normalmente, são grandes, trabalhosas e incompreensíveis.

Podemos tomar como exemplo a geração da folha de pagamento de uma empresa. Para simplificar, vamos considerar que são dados de entrada o percentual de desconto aplicado igualmente a todos os funcionários e para cada funcionário serão fornecidos nome, salário contratual e vantagens.

Como este exemplo ainda é muito simples, podemos resolvê-lo de forma legível, utilizando uma matriz de float. Apesar de correta e legível, não representaria claramente o registro de um funcionário, pois o nome deste estaria em um vetor de strings e seus dados salariais (salário contratual, vantagens, desconto, salário líquido) estaria em uma matriz. A unidade de funcionário estaria separada em duas estruturas homogêneas.

Podemos notar que armazenar mais um dado para um funcionário pode não ser tão fácil. Por exemplo, quisermos guardar a data de admissão do funcionário. Como armazenaríamos esta data? Não podemos usar a matriz de foat. Usar um vetor de strings para guardar as datas é uma solução fraca. Além disso, teríamos agora três matrizes para armazenar os funcionários. Imagine se tivermos que armazenar dados como férias, dependentes, CPF, carteira de trabalho etc! Não conseguiríamos fazer um programa legível com um quantidade tão grande de vetores e matrizes. Com a evolução do programa, chegaríamos a um ponto que seria impossível entendê-lo.

As linguagens de programação estruturada possuem um recurso conhecido como registro, também conhecido como estrutura. Os registros são tipos definidos pelo programador, capazes de armazenar um conjunto heterogêneo de dados. Isto significa que podemos definir um registro de funcionário capaz de armazenar em apenas uma variável todos os dados necessários. A definição de registros em C é feita da seguinte forma:

```
struct nome_da_estrutura{
    tipo1 nome_campo1;
    tipo2 nome_campo2;
    . . .
    tipoN nome_campoN;
};
```

Cada dado armazenado em uma estrutura é conhecido como campo. Vamos definir um registro de funcionário:

```
struct funcionario{
  char nome[31];
  float sal_contr, vant, desc, sal_liq;
};
```

Para declarar uma variável do tipo registro devemos utilizar a palavra struct. Para acessar cada campo do registro utilizamos a notação de ponto (nome_variavel.nome_campo). Observe o exemplo do algoritmo 20.

```
Nome:  Joao de Deus
Salario Contratual:  500.00
Vantagens:  80.00
```

```
Descontos:  50.00
Salario Liquido:  530.00
```

Algoritmo 20: Exemplo com registro

```
1  int main(){
2    struct funcionario func;
3    float desc_perc;
4    desc_perc = 0.10f;
5    strcpy(func.nome, "Joao de Deus");
6    func.sal_contr = 500;
7    func.vant = 80;
8    func.desc = func.sal_contr * desc_perc;
9    func.sal_liq = func.sal_contr + func.vant - func.desc;
10   printf("Nome: %s\nSalario Contratural: %.2f\n Vantagens: %.2f\n\
11   Descontos: %.2f\n\
12   Salario Liquido: %.2f\n",
13   func.nome, func.sal_contr,
14   func.vant, func.desc, func.sal_liq);
15   return 0;
16 }
```

5.1 Vetores de registros

São raras as situações em que faremos um programa que utilize apenas uma variável do tipo registro. Normalmente utilizaremos uma quantidade de variáveis deste tipo, indefinida em tempo de programação. Uma forma de fazermos isso é utilizar um vetor de registros. O vetor de registros funciona de forma idêntica ao vetor de tipos primitivos, aqueles oferecidos pela linguagem. Veja o exemplo abaixo:

Algoritmo 21: Vetores de registros

```
1  int main(){
2    struct funcionario func[TAM];
3    float desc_perc;
4    int quant_func = TAM;
5    printf("Desconto (%%): ");
6    scanf("%f", &desc_perc);
7    ler_dados_reg_func(func, quant_func);
8    calcular_folha_reg_func(func, quant_func, desc_perc);
9    imprimir_folha_reg_func(func, quant_func);
10   return 0;
11 }
```

Onde as funções de ler, calcular e imprimir poderiam ser implementadas da seguinte forma:

Caso não saibamos a quantidade de funcionários em tempo de programação, podemos implementar um programa utilizando alocação dinâmica de vetores de registros, como mostra o exemplo abaixo:

As funções para ler calcular e imprimir continuam idênticas.

5.2 Definição de tipos

A declaração de registros em C é um pouco ilegível. Para tornar nossos programas um pouco mais legíveis, podemos utilizar a definição de tipos da linguagem C.

```
typedef float Real;
```

O código acima nos permite declarar variáveis do tipo Real.

De forma análoga podemos definir um tipo Funcionario:

Algoritmo 22: Leitura de dados para um registro

```
1  void ler_dados_reg_func(struct funcionario *func, int quant_func){
2    int i;
3    for(i=0; i<quant_func; i++){
4      printf("Nome %d: ", i+1);
5      scanf(" %30[^\n]", func[i].nome);
6      printf("Salario %d: ", i+1);
7      scanf("%f", &(func[i].sal_contr));
8      printf("Vantagem %d: ", i+1);
9      scanf("%f", &(func[i].vant));
10   }
11 }
```

Algoritmo 23: Calcula folha de pagamento

```
1  void calcular_folha_reg_func(struct funcionario *func, int quant_func, float
       desc_perc){
2    int i;
3    for(i=0; i<quant_func; i++) {
4      func[i].desc = func[i].sal_contr * desc_perc;
5      func[i].sal_liq = func[i].sal_contr + func[i].vant - func[i].desc;
6    }
7  }
```

Algoritmo 24: Imprime folha de pagamento

```
1  void imprimir_folha_reg_func(struct funcionario *func, int quant_func){
2    int i;
3    printf("\n%30s\tsal_contr\tvantagens\tdesconto\tsal_liqui\n", "Nome");
4    for(i=0; i<quant_func; i++) {
5      printf("%30s\t%9.2f\t%9.2f\t%9.2f\t%9.2f\n",func[i].nome, func[i].
           sal_contr, func[i].vant, func[i].desc, func[i].sal_liq);
6    }
7  }
```

Algoritmo 25: Programa com registros

```
1  int main(){
2    struct funcionario *func;
3    float desc_perc;
4    int quant_func;
5    printf("Quantidade de funcionarios: ");
6    scanf("%d", &quant_func);
7    func = malloc(sizeof(struct funcionario)*quant_func);
8    printf("Desconto (%%): ");
9    scanf("%f", &desc_perc);
10   ler_dados_reg_func(func, quant_func);
11   calcular_folha_reg_func(func, quant_func,desc_perc);
12   imprimir_folha_reg_func(func, quant_func);
13   return 0;
14 }
```

Algoritmo 26: Exemplo com registro

```
1 int main(){
2   Real numero;
3   printf("Digite um numero real: ");
4   scanf("%f", &numero);
5   printf("O numero e %.2f.\n");
6   return 0;
7 }
```

```
typedef struct funcionario Funcionario;
```

Podemos também definir um tipo como um ponteiro para outro tipo ou estrutura.

```
typedef int *PInteiro;
typedef struct funcionario *PFuncionario;
```

Ponteiros para estruturas Podemos declarar mais de um nome em uma única declaração typedef.

```
typedef struct estrutura Estrutura, *PEstrutura;
```

Cada declaração abaixo cria uma estrutura ao iniciar o bloco onde ela foi declarada:

```
struct estrutura estrutura1;
Estrutura estrutura2;
```

Não é incomum precisarmos de declarar apenas ponteiros para estruturas.

```
struct estrutura *ponteiro_estrutura1;
Estrutura *ponteiro_estrutura2;
PEstrutura ponteiro_estrutura3;
```

Nos três casos precisamos alocar a estrutura:

```
ponteiro_estrutura = malloc(sizeof(struct estrutura));
```

ou ainda

```
ponteiro_estrutura = malloc(sizeof(Estrutura));
```

Quando usamos ponteiros para estruturas podemos acessar os campos da estrutura de duas formas:

```
PEstrutura ponteiro_estrutura;
tipo_campo_estrutura campo_estrutura;
ponteiro\_estrutura = malloc(sizeof(Estrutura));
campo_estrutura = (*ponteiro_estrutura).campo_estrutura;
```

ou

```
campo_estrutura = ponteiro_estrutura->campo_estrutura;
```

A notação '->' é mais comum.

Exercícios

Exercício 1

Considere a struct ponto, que representa um ponto em um plano cartesiano:

```
struct ponto {
 float x;
 float y;
};
```

1. Faça uma função para ler um ponto pelo terminal.

2. Faça uma função que imprima um ponto.
3. Faça uma função que determine a distância entre dois pontos.
4. Faça uma função que calcule a área de um triângulo através dos seus três vértices passados por pontos.
5. Faça uma função que calcule a área de um retângulo através dos pontos de seu canto superior esquerdo e inferior direito.
6. Considere que um polígono qualquer é representado por um vetor de pontos. Faça um função que calcule a área desse polígono.

Exercício 2

Desejamos armazenar uma tabela com dados de alunos. Podemos organizar os dados dos alunos em um vetor. Para cada aluno, vamos supor que sejam necessárias as seguintes informações:

nome: cadeia com até 80 caracteres
matricula: número inteiro
endereço: cadeia com até 120 caracteres
telefone: cadeia com até 20 caracteres

1. Faça uma estrutura para representar os dados de um aluno.
2. Faça uma função para alocar uma variável aluno.
3. Faça uma função para preencher os dados de uma variável aluno, já aloca, via terminal.
4. Faça uma função que libere um variável aluno alocada dinamicamente.
5. Faça uma função que imprima um aluno.
6. Faça um programa que leia e liste os dados de uma quantidade indeterminada de alunos, sem alocar alunos desnecessariamente. Utilize um vetor de tamanho fixo.

Apêndice A

Reuso de programas

Sabemos programar não é uma tarefa fácil. Por isso, sempre que podemos, devemos reutilizar os códigos que já fizemos. Porém, reutilizar código não é copiar e colar um código pronto em uma nova aplicação. A ideia de um padrão de desenvolvimento copy&paste é inconcebível. Imagine se quisermos alterar uma implementação depois desta ser copiada e colada mil vezes! Será que conseguiríamos alterar todas as cópias da implementação? Ou deveríamos adaptar os novos códigos a implementação antiga, tornando-a imutável? Claro que a resposta correta as estas duas perguntas é NÃO. Então, se não podemos copiar e colar um código, como devemos programar para que a alteração de uma implementação ocorra em apenas um lugar?

A.1 Dividindo programas

A linguagem C nos permite dividir nossos programas em mais de um arquivo fonte. Isso nos permite utilizar um trecho de código em mais de um programa. Suponha que precisamos de fazer um programa que calcule as raízes de uma equação do segundo grau. Podemos implementar a fórmula de Báskara diretamente no programa, mas se implementarmos a fórmula de Báskara em um arquivo separado, poderemos fazer vários programas que usam esta implementação.

Algoritmo 27: baskara.c

```
1  #include <math.h>
2  float delta(float a, float b, float c) {
3    return b*b - 4*a*c;
4  }
5  /* Coloca o valor das raizes nas variaveis x1 e x2, se existirem.
6  ** Retorna a quantidade de raizes encontradas. */
7  int baskara(float a, float b, float c, float *x1, float *x2) {
8    float d = delta(a, b, c);
9    /* Se o delta negativo a equacao nao possui raizes */
10   if(d < 0)
11     return 0;
12   /* Se o delta for zero so existe uma raiz */
13   if(d == 0){
14     *x1 = *x2 = ((-1)*b)/(2*a);
15     return 1;
16   }
17   /* Se o delta for positivo existem duas raizes reais */
18   *x1 = ((-1)*b+sqrt(d))/(2*a);
19   *x2 = ((-1)*b-sqrt(d))/(2*a);
20   return 2;
21 }
```

Note que incluímos a biblioteca *math.h*, pois utilizamos a função *sqrt*. Porém, não incluímos a bilbioteca *stdio.h*, já que não utilizamos nenhuma função de entrada e saída padrão. Isso nos permite utilizar esta

implementação da fórmula de Báskara em vários programas, que utilizam diferentes interfaces com o usuário, ou até mesmo em programas que necessitam da implementação desta fórmula e o usuário sequer sabe que esta fórmula esta sendo utilizada. Para utilizarmos as funções implementadas em *baskara.c* basta incluirmos (*include*) este arquivo no código fonte onde ele é necessário.

Algoritmo 28: eq2grau.c

```
1  #include "baskara.c"
2  #include <stdio.h>
3  int main() {
4    float a, b, c, x1, x2;
5    int qraizes;
6    printf("Digite os valores de a, b e c: ");
7    scanf("%f %f %f", &a, &b, &c);
8    qraizes = baskara(a, b, c, &x1, &x2);
9    if(a == 0)
10     printf("Para ser equacao do 2o. grau 'a' deve ser diferente de zero.\n");
11   else
12     if(qraizes == 0)
13       printf("Esta equacao nao possui raizes reais.\n");
14     else
15       if(qraizes == 1)
16         printf("Esta equacao possui apenas uma raiz real: x = %f\n", x1);
17       else
18   printf("Esta equacao possui duas raizes reais: x1 = %f e x2 = %f\n", x1, x2)
       ;
19   return 0;
20 }
```

Note que o arquivo *baskara.c* esta entre aspas e não entre < e >. O arquivo *baskara.c* deve estar no mesmo diretório do arquivo *eq2grau.c*. Agora que precisamos de interação com o usuário, podemos incluir a biblioteca *stdio.h*. Se refizermos este programa utilizando uma interface gráfica, podemos ainda utilizar o arquivo *baskara.c*. O comando do gcc para compilar este programa pode ser: `gcc eq2grau.c -o eq2grau.bin -Wall -lm`.

A.2 Escondendo o código fonte

Na maioria das vezes precisamos de saber apenas para que serve uma função, quais são seus parâmetros e seus possíveis valores de retorno. Por outro lado, existem situações em que queremos compartilhar a utilização dos nossos códigos mas queremos esconder a nossa implementação. Para isso podemos dividir nossas implementações em dois arquivos: Um com a interface das funções (quais seus parâmetros e possíveis valores de retorno) e outro com a implementação propriamente dita. Os arquivos com interfaces de programação de aplicações (API - Application Programming Interface) possuem a extensão '.h', que significa Header, ou cabeçalho.

Algoritmo 29: baskara.h

```
1  /* Coloca o valor das raizes nas variaveis x1 e x2, se existirem.
2  ** Retorna a quantidade de raizes encontradas. */
3  int baskara(float a, float b, float c, float *x1, float *x2);
```

Note que só colocamos a assinatura da função baskara. Quem utilizar nossa implementação não precisa saber que usamos uma função auxiliar delta para calcular as raízes da equação, como mostra o algoritmo 30.

A função delta foi implementada mas não foi declarada no arquivo de cabeçalho. Isto significa que podemos utilizar função delta apenas no arquivo baskara.c. Nós incluímos o arquivo baskara.h no arquivo baskara.c. Isto permite que utilizemos, dentro do arquivo .c, uma função declarada no arquivo .h, acima de

Algoritmo 30: baskara.c

```
1  #include "baskara.h"
2  #include <math.h>
3  float delta(float a, float b, float c) {
4    return b*b - 4*a*c;
5  }
6  int baskara(float a, float b, float c, float *x1, float *x2) {
7    float d = delta(a, b, c);
8    if(d < 0)
9      return 0;
10   if(d == 0){
11     *x1 = *x2 = ((-1)*b)/(2*a);
12     return 1;
13   }
14   *x1 = ((-1)*b+sqrt(d))/(2*a);
15   *x2 = ((-1)*b-sqrt(d))/(2*a);
16   return 2;
17 }
```

sua implementação. Além disso, a inclusão do arquivo .h no seu respectivo .c, garante que as assinaturas e os tipos de retorno das funções estão iguais. Podemos agora criar um módulo compilado de baskara. Este modulo possui a extenção .o (object) e é utilizado em conjunto com o arquivo .h. O comando do gcc para compilar um arquivo baskara.c em baskara.o é: `gcc baskara.c -c`. Este comando gera o arquivo baskara.o no mesmo diretório que baskara.c. Depois da compilação em .o não precisamos mais do arquivo .c. Para utilizarmos esta biblioteca que acabamos de criar, podemos implementar nosso programa da conforme algoritmo 31.

Algoritmo 31: eq2grau.c

```
1  #include "baskara.h"
2  #include <stdio.h>
3  int main(){
4    float a, b, c, x1, x2;
5    int qraizes;
6    printf("Digite os valores de a, b e c: ");
7    scanf("%f %f %f", &a, &b, &c);
8    qraizes = baskara(a, b, c, &x1, &x2);
9    if(a == 0)
10     printf("Para ser equacao do 2o. grau 'a' deve ser diferente de zero.\n");
11   else if(qraizes == 0)
12       printf("Esta equacao nao possui raizes reais.\n");
13        else if(qraizes == 1)
14          printf("Esta equacao possui apenas uma raiz real: x = %f\n", x1);
15             else
16     printf("Esta equacao possui duas raizes reais: x1 = %f e x2 = %f\n", x1,
           x2);
17   return 0;
18 }
```

Note que incluímos o arquivo *baskara.h* e não *baskara.c*. O comando *gcc* para compilar este programa agora seria: `gcc eq2grau.c -o eq2grau.bin "baskara.o" -lm`. Da mesma forma que precisamos de *-lm* para carregar o módulo da *math.h*, também precisamos no *baskara.o* para utilizar o *baskara.h*. Os módulos locais, como baskara.o, devem ser carregados entre aspas.

Referências Bibliográficas

E. W. Dijkstra. *A short introduction to the art of programming*, volume 4. Technische Hogeschool Eindhoven Eindhoven, 1971.

float.h. <cfloat> (float.h). `http://www.cplusplus.com/reference/cfloat`, 2019. Acessado: 22 de maio de 2019.

GNU Project. GCC, the GNU Compiler Collection. `http://www.gnu.org/software/gcc/gcc.html`, 2019. Acessado: 22 de maio de 2019.

gnuplot. gnuplot. `http://www.gnuplot.info`, 2019. Acessado: 22 de maio de 2019.

`https://en.wikipedia.org/wiki/C_data_types`. C data types, 2019. Acessado: 22 de maio de 2019.

`http://www.cplusplus.com/reference/cstdio/printf`. function printf, 2019. Acessado: 22 de maio de 2019.