Programação em C

Francisco de Assis Boldt

7 de junho de 2018

Sumário

Capítulo 1

Programas em C

O menor programa completo [1] em C é apresentado no algoritmo 1.

Algoritmo 1: faznada.c

```
1  int main(){
2      return 0;
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 (;).

A palavra `main` indica que esta é a função principal do programa. No caso do algoritmo 1 é a única função do programa. Mas, um arquivo fonte escrito em C pode conter várias funções. Porém, a função `main` será a primeira a ser chamada pelo SO quando um programa escrito em C for executável. 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.

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`, como pode ser visto no algoritmo 2. 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.

Algoritmo 2: hello.c

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

[1] 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.

1.2 Variáveis

Programas de computadores executam essencialmente operações matemáticas. Operações como soma podem ser executadas, como mostrado no algoritmo 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 analizar o código para descobrir.

Algoritmo 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 }
```

Podemos notar que se precisarmos alterar este programa, por exemplo trocando de 5 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. Então, poderíamos fazer este programa de um forma um pouco mais reutilizável, como mostra o algoritmo 4. Com este algoritmo, caso queiramos mudar de 5 para 8, basta alterarmos as linhas 5 e 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.

Algoritmo 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.

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 algoritmo 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.

Algoritmo 5: 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  /* Programa que chama o procedimento hello() tres vezes. */
7  int main() {
8      hello();
9      hello();
10     hello();
11     return 0;
12 }
```

Em C, a diferença entre função e procedimento está no retorno da função. No exemplo do algoritmo 5, a declaração do subprograma `hello` inicia com a palavra reservada `void`, indicando que este subprograma não retorna valor algum e, portanto, é um procedimento.

2.2 Parâmetros

Programas são mais versáteis quando geram saídas diferentes dependendo da entrada. Por exemplo, o procedimento `hello` no algoritmo 5 imprime sempre a mesma frase, o que o torna muito limitado. Muito mais interessante é o procedimento `imprime_soma` apresentado no algoritmo 6. Nota-se que, depois depois de criado, o subprograma pode ser reutilizado quantas vezes for necessário. Ele gera resultados diferentes dependendo dos parâmetros passados.

Ao definir um subprograma, seja ele uma função ou um procedimento, devemos incluir a lista de parâmetros. No caso do procedimento `hello` no algoritmo 5, 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.

Algoritmo 6: 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 }
```

Exercícios

1. Implemente o procedimento `imprime_subtracao`.

2. Implemente o procedimento `imprime_multiplicacao`.

3. Implemente o procedimento `imprime_divisao`.

4. Teste os procedimentos implementados em uma só execução dentro da função `main`.

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. Em linguagem C esta implementação pode ser feita como mostra o algoritmo 7.

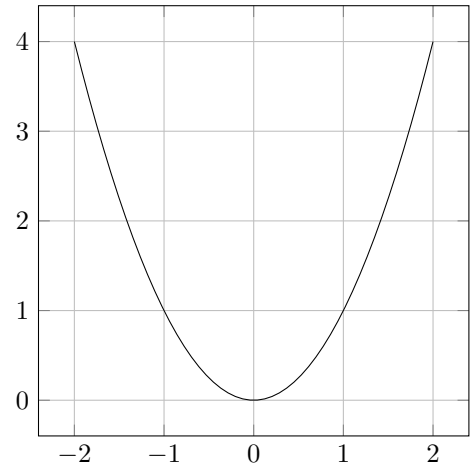


Figura 2.1: $f(x) = x^2$

O algoritmo 7 apresenta algumas novidades. A primeira delas é a palavra reservada `float`, que aparece duas vezes na linha 2 e uma vez na linha 6. A palavra `float` é uma das palavras reservadas que diz ao SO que uma variável, um parâmetro ou o retorno de uma função é um número real, não um número inteiro como quando se usa a palavra `int`. Números inteiros e reais são processados em diferentes partes do processador do computador. Algumas linguagens fazem esta distinção automaticamente, mas este não é o caso da linguagem C. Então, o parâmetro `x` da função `quadrado` é usado para gerar o retorno desta função, que também é um valor do tipo `float`.

Algoritmo 7: quadrado.c

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

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 comummente necessárias. Veja o exemplo apresentado no algoritmo 8.

Algoritmo 8: coseno.c

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

Neste programa nós usamos a função `cos`[2] da bilioteca `math.h` para calcular o coseno (figura 2.2) 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.

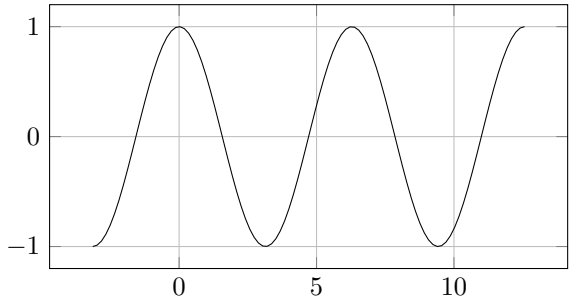


Figura 2.2: $f(x) = cos(x)$

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

Exercícios

1. Implemente uma função para converter uma polegada para milímetros. A formula de conversão é 25,4 mm = 1 polegada.

2. 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.

3. Implemente uma função para converter uma temperatura em graus Fahrenheit para Celsius.

4. Implemente uma função que receba dois números positivos representando os comprimentos dos lados de um retângulo e retorne a área desse retângulo.

5. 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.

6. Implemente uma função que receba dois números positivos representando os lados de um retângulo e retorne seu perímetro.

7. 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 4 e 6.

8. 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`.

9. 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.

10. 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.

2.4 Dividir para conquistar - Blocos de construção

Funções contínuas são normalmente mais fáceis de implementar do que funções descontínuas. Os exercícios resolvidos que seguem mostram algumas formas de lidar com descontinuidade de funções.

Criando os blocos

Função rampa

Analise a função da figura 2.3 e a implemente em linguagem C. [3]

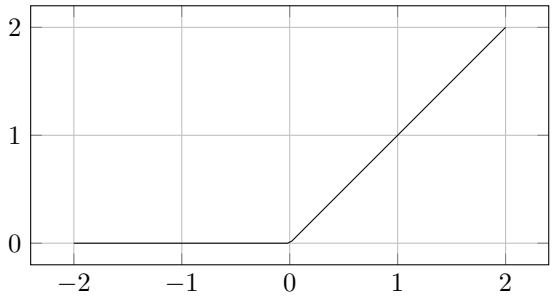


Figura 2.3: $f(x) = rampa(x)$

[3] Dica: use a função `fabs` da biblioteca `math.h`.

Solução:

A função da figura 2.3 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 um função descontínua. Vamos 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 2.4, onde a linha contínua representa a função identidade ($f_1(x) = x$), a linha tracejada representa a função módulo ($f_2(x) = |x|$) e a linha pontilhada representa a soma dessas duas funções ($f_3(x) = f_1(x) + f_2(x)$). 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.

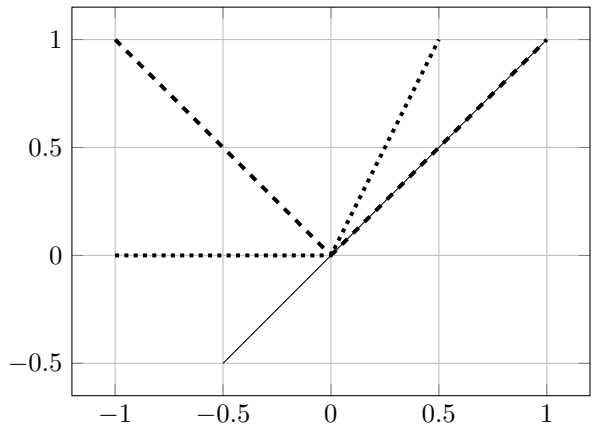


Figura 2.4: $f_1(x) = x$, $f_2(x) = |x|$, $f_3(x) = f_1(x) + f_2(x)$.

Como a função para números positivos é $f(x) = 2x$, se aplicarmos a função inversa ($f^{-1}(x) = \frac{x}{2}$) 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 9.

Algoritmo 9: rampa.c

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

Este algoritmo possui algumas partes que precisam ser explicadas. Na linha 4 foi usada a função `fabs` da biblioteca `math.h`. A biblioteca `math.h` também possui a função `abs`, mas esta só funciona para números inteiros (`int`). Para números reais (`float`), precisamos usar a função `fabs`. Nas linhas 7, 8 e 9, colocamos os números o `.0` no final de alguns números. Em C, por padrão, constantes sem o `.0` são consideradas números inteiros. Então, dependendo do compilador ou arquitetura que você usa, o `%f` do `printf` pode não entender a constante inteira e imprimir zero ao invés do número desejado. Por outro lado, quando passamos os valores para a função `rampa` como parâmetros, o compilador já entende que este é um número real pois já foi declarado como tal na linha 3. O retorno da função `rampa` é um número real e portanto compreendido pela função `printf`.

Rampa invertida

Analise a função na figura 2.5 e implemente-a em C. Siga o processo de desenvolvimento mostrado no exercício resolvido anterior. A solução desse exercício é basicamente aquela apresentada no algoritmo 9, alterando as linhas 3 e 4, como mostra o algoritmo 10. A função pode ser escrita como $f(x) = \frac{x-|x|}{2}$.

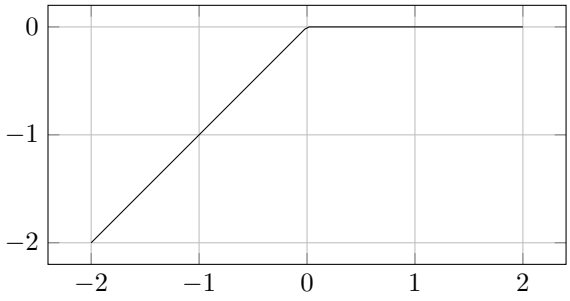


Figura 2.5: $f(x) = rampainv(x)$

Algoritmo 10: rampainv.c

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

Rampa invertida com máximo 1

A função da figura 2.6 é muito parecida com a função da figura 2.5. Use a função `rampainv` para implementar a função da figura 2.6. Uma possível solução é apresentada no algoritmo 11. A função também pode ser escrita como $f(x) = \frac{(x-1)-|(x-1)|}{2} + 1$, conforme mostra a equação 2.1.

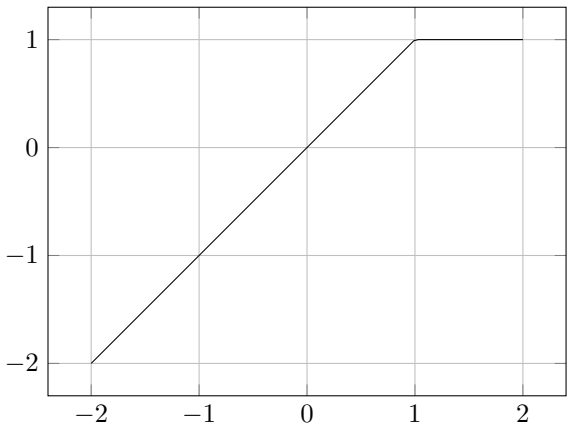


Figura 2.6: $f(x) = rampainvmax1(x)$

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

Algoritmo 11: rampainvmax1.c

```
1  #include <stdio.h>
2  #include <math.h>
3  float rampainv(float x) {
4    return (x-fabs(x))/2;
5  }
6  float rampainvmax1(float x) {
7    return rampainv(x-1)+1;
8  }
9  int main() {
10   printf("f(%f) = %f\n", -1.0, rampainvmax1min0(-1));
11   printf("f(%f) = %f\n", 0.0, rampainvmax1min0(0));
12   printf("f(%f) = %f\n", 1.0, rampainvmax1min0(1));
13   printf("f(%f) = %f\n", 2.0, rampainvmax1min0(2));
14   return 0;
15 }
```

Rampa invertida com máximo 1 e mínimo 0

Implemente a função da figura 2.7 usando as funções `rampa` e `rampinv`. Baseie-se na implementação da função `rampainvmax1` apresentada no algoritmo 11.

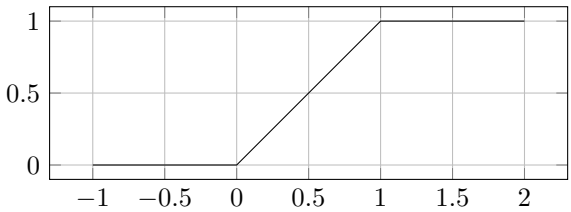


Figura 2.7: $f(x) = rampainvmax1min0(x)$

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 algoritmo 12 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 $h(x)+1$. Então, $f(x) = g(h(x)+1)$.

A função expandida é apresentada na equação 2.2.

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

Algoritmo 12: rampainvmax1min0.c

```
1  #include <stdio.h>
2  #include <math.h>
3  float rampa(float x) {
4    return (x+fabs(x))/2;
5  }
6  float rampainv(float x) {
7    return (x-fabs(x))/2;
8  }
9  float rampainvmax1min0(float x) {
10   return rampa(rampainv(x-1)+1);
11 }
12 int main() {
13   printf("f(%f) = %f\n", -1.0, rampainvmax1min0(-1));
14   printf("f(%f) = %f\n", 0.0, rampainvmax1min0(0));
15   printf("f(%f) = %f\n", 0.5, rampainvmax1min0(0.5));
16   printf("f(%f) = %f\n", 1.0, rampainvmax1min0(1));
17   printf("f(%f) = %f\n", 2.0, rampainvmax1min0(2));
18   return 0;
19 }
```

Rampa íngreme

Enquanto a rampa da função na figura 2.7 tem 45° de inclinação a rampa da função na figura 2.8 tem 60° de inclinação. Implemente a função da figura 2.8.

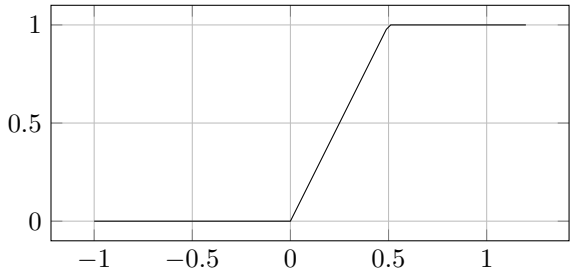


Figura 2.8: $f(x) = rampaingreme(x)$

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.

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

Algoritmo 13: rampaingreme.c

```
1  #include <stdio.h>
2  #include <math.h>
3  float rampa(float x) {
4    return (x+fabs(x))/2;
5  }
6  float rampainv(float x) {
7    return (x-fabs(x))/2;
8  }
9  float rampaingreme(float x) {
10   const float c = 0.5;
11   return rampa(rampainv(x-c)+c)/c;
12 }
13 int main() {
14   printf("f(%f) = %f\n", -1.0, rampaingreme(-1));
15   printf("f(%f) = %f\n", 0.0, rampaingreme(0));
16   printf("f(%f) = %f\n", 0.25, rampaingreme(0.25));
17   printf("f(%f) = %f\n", 0.5, rampaingreme(0.5));
18   printf("f(%f) = %f\n", 1.0, rampaingreme(1));
19   return 0;
20 }
```

A linha 10 do algoritmo 13 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 [4]. 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 que a próxima função faz. Ela pode ser usada para fazer vários algoritmos importantes na programação de computadores.

[4] No caso do algoritmo 13 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 palavras `const` garante que o valor dentro de uma constante não será alterado.

Função degrau

Baseado na função `rampaingreme` do algoritmo 13, implemente a função `degrau` mostrada na figura 2.9

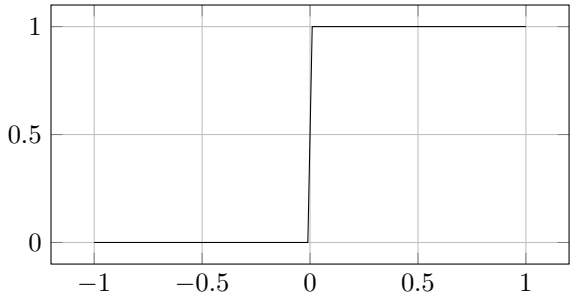


Figura 2.9: $f(x) = degrau(x)$

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

$$\begin{aligned} c &= 0.00000000000000000001 \\ g(x) &= \frac{x+|x|}{2} \\ h(x) &= \frac{x-|x|}{2} \\ f(x) &= \frac{g(h(x-c)+c)}{c} \\ &= \frac{g(\frac{x-c-|x-c|}{2}+c)}{c} \\ &= \frac{\frac{x-c-|x-c|}{2}+c+|\frac{x-c-|x-c|}{2}+c|}{2} \\ &\qquad c \end{aligned} \tag{2.4}$$

Algoritmo 14: degrau.c

```
1  #include <stdio.h>
2  #include <math.h>
3  float rampa(float x) {
4    return (x+fabs(x))/2;
5  }
6  float rampainv(float x) {
7    return (x-fabs(x))/2;
8  }
9  float degrau(float x) {
10   const float c = 0.00000000000000000001;
11   return rampa(rampainv(x-c)+c)/c;
12 }
13 int main() {
14   printf("f(%f) = %f\n", -0.000001, degrau(-0.000001));
15   printf("f(%f) = %f\n", 0.0, degrau(0));
16   printf("f(%f) = %f\n", 0.000001, degrau(0.000001));
17   return 0;
18 }
```

A função é degrau é muito útil para resolver problemas. O algoritmos a seguir mostram algumas aplicações dela.

Usando os blocos

1. Utilizando a função degrau apresentada no algoritmo 14, 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

Algoritmo 15: 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 = 0.00000000000000000001;
10   return rampa(rampainv(x-c)+c)/c;
11 }
12 float maior(float x, float y) {
13   return degrau(x-y);
14 }
15 float diferente(float x, float y) {
16   return maior(x,y)+maior(y,x);
17 }
18 float igual(float x, float y) {
19   return maior(1,diferente(x,y));
20 }
21 float maior2(float x, float y){
22   return x*maior(x,y)+y*maior(y,x)+x*igual(x,y);
23 }
24 float maior3(float x1, float x2, float x3) {
25   return maior2(maior2(x1,x2),x3);
26 }
27 float menor2(float x, float y){
28   return x*maior(y,x)+y*maior(x,y)+x*igual(x,y);
29 }
30 void decrescente2(float x1, float x2) {
31   printf("%f %f\n", maior2(x1, x2), menor2(x1, x2));
32 }
```

A seguir a explicação das soluções apresentadas no algoritmo 15

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^{-20} a função `degrau` retorna o valor de entrada. Então, neste intervalo, a função maior vai falhar. Quando `x` for pelo menos 10^{-20} 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 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 mostram que utilizar a função `degrau` para trabalhar com funções descontínuas pode ser uma tarefa árdua. Para facilitar a vida dos programadores, a maioria das linguagens de programação, incluindo C, oferecem operadores lógicos. O algoritmo 16 mostra como a função degrau pode ser implementada usando o operador lógico > (maior) ao invés das funções `rampa` e `rampainv`.

Algoritmo 16: degrau_relacional.c

```
1  #include <stdio.h>
2  float degrau(float x) {
3    return x>0;
4  }
5  int main() {
6    printf("f(%f) = %f\n", -0.00000000001, degrau(-0.00000000001));
7    printf("f(%f) = %f\n", 0.0, degrau(0));
8    printf("f(%f) = %f\n", 0.00000000001, degrau(0.00000000001));
9    return 0;
10 }
```

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 algoritmo 17 mostra como fica a função `maior2` implementada usando operadores lógicos.

Algoritmo 17: maior2.c

```
1  #include <stdio.h>
2  float maior2(float x, float y) {
3    return x*(x>y) + y*(y>x) + x*(x==y);
4  }
5  int main() {
6    printf("%f > %f ==> %f\n", 5.0, 4.0, maior2(5,4));
7    printf("%f > %f ==> %f\n", 4.0, 4.0, maior2(4,4));
8    printf("%f > %f ==> %f\n", 3.0, 4.0, maior2(3,4));
9    return 0;
10 }
```

Observe que a implementação da função `maior2` no algoritmo 17 pode ser feita apenas substituindo a função `maior` pelo operador > e a função `igual` pelo operador ==.

Implemente, 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 algoritmo 18. Nele foram criadas duas variáveis para receber o menor e o maior valor. Nada impede de se fazer duas funções para retornar o menor e o maior de dois valores.

Algoritmo 18: crescente2.c

```
1  #include <stdio.h>
2  void crescente2(float x1, float x2) {
3    float menor = x1*(x1<=x2)+x2*(x1>x2);
4    float maior = x1*(x1>=x2)+x2*(x1<x2);
5    printf("%f %f\n", menor, maior);
6  }
7  int main(){
8    printf("crescente2(3,2); ");
9    crescente2(3,2);
10   printf("crescente2(3,4); ");
11   crescente2(3,4);
12   return 0;
13 }
```

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 algoritmo 18, 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`.

Algoritmo 19: crescente3.c

```
1  #include <stdio.h>
2  void crescente3(float x1, float x2, float x3) {
3     float menor = x1*(x1==x2)*(x1==x3)+
4                   x1*(x1<x2)*(x1<x3)+x1*(x1<x2)*(x1==x3)+x1*(x1==x2)*(x1<x3)+
5                   x2*(x2<x1)*(x2<x3)+x2*(x2<x1)*(x2==x3)+x3*(x3<x1)*(x3<x2);
6     float maior = x1*(x1==x2)*(x1==x3)+
7                   x1*(x1>x2)*(x1>x3)+x1*(x1>x2)*(x1==x3)+x1*(x1==x2)*(x1>x3)+
8                   x2*(x2>x1)*(x2>x3)+x2*(x2>x1)*(x2==x3)+x3*(x3>x1)*(x3>x2);
9     float medio = x1*(x1==x2)*(x1==x3)+
10                  x1*(x1>x2)*(x1<x3)+x1*(x1<x2)*(x1>x3)+x2*(x2>x1)*(x2<x3)+
11                  x2*(x2<x1)*(x2>x3)+x3*(x3>x1)*(x3<x2)+x3*(x3<x1)*(x3>x2)+
12                  x1*(x1==x2)*(x1<x3)+x1*(x1==x3)*(x1<x2)+x1*(x1==x2)*(x1>x3)+
13                  x1*(x1==x3)*(x1>x2)+x2*(x2==x3)*(x3<x2)+x2*(x2==x3)*(x3>x2);
14    printf("%f %f %f\n", menor, medio, maior);
15 }
16 int main(){
17    printf("crescente3(3,2,1); ");
18    crescente3(3,2,1);
19    return 0;
20 }
```

O algoritmo 19 mostra que este 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 algoritmo 20 mostra uma ideia que NÃO FUNCIONA.

Algoritmo 20: 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 }
```

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. 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`. Os parâmetros do procedimento possuem apenas cópias dos valores das variáveis externas. Portanto, não adianta trocar os valores dentro do procedimento, porque eles não serão alterados fora dele.

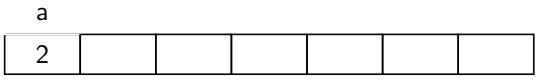
Entretanto, é possível em C, alterar valores que existem fora de um procedimento, usando a técnica de passagem de parâmetros por referência. Esta técnica consiste em passar como parâmetro o endereço de memória das variáveis que serão alteradas. Um exemplo de como se usar a passagem de parâmetros por referência é mostrado no algoritmo 21. O procedimento `atribui10` coloca o valor 10 à posição de memória passada por referência.

Algoritmo 21: atribui10.c

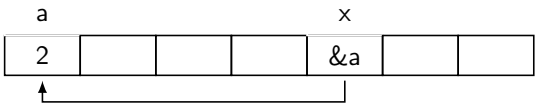
```
1  #include <stdio.h>
2  void atribui10(float *x) {
3    *x = 10;
4  }
5  int main() {
6    float a=2;
7    printf("a=%f\n", a);
8    atribui10(&a);
9    printf("a=%f\n", a);
10   return 0;
11 }
```

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 9: `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.

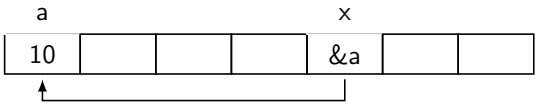
A figura 2.10 ilustra a memória do computador durante a execução do algoritmo 21. Quando o programa se inicia, o SO reserva uma quantidade de memória para o programa, que está inicialmente livre. Na linha 6, o programa define uma posição de memória para guardar o valor da variável `a` e atribui o valor 2 a ela (sub-figura 2.10a). 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`) (sub-figura 2.10b). A linha 3 altera o valor da posição de memória apontada por `x`, fazendo com que o valor de `a` mude para 10 (sub-figura 2.10c). Inicialmente, esta parte é um pouco confusa, porque parece que estamos atribuindo um valor à variável `x` e estamos alterando o valor da variável `a`. Porém, o que estamos fazendo na verdade, é usar o valor armazenado em `x` para saber onde `a` está na memória, e só então alterar o valor de `a`.



(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 2.10: Memória do computador durante a execução do algoritmo 21.

Implemente um procedimento que receba dois parâmetros e troque seus valores nas variáveis externas.

Algoritmo 22: 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 algoritmo 22 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.

Exercícios

1. Inspirando-se no algoritmo 22, corrija o algoritmo 20 de forma que ele funcione.
2. Utilizando o procedimento `crescente2correto`, implemente um procedimento 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.

Soluções

A função `main` do algoritmo 23 é muito parecida com a função `main` do algoritmo 22. 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.

Algoritmo 23: 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 algoritmo 24. Isso é possível graças à passagem por referência usando ponteiros.

Algoritmo 24: 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 algoritmo 19. 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 observar 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.

Algoritmo 25: 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 25 é análogo ao do algoritmo 24. Nota-se portanto, que a implementação 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 algoritmo 26 mostra um exemplo da utilização da função `scanf`.

Algoritmo 26: 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 do algoritmo 26 a função `scanf` é usada. Note que a sintaxe desta função é muito parecida com a sintaze 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 espeço 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. 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.
3. 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.
4. 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.

Algoritmo 27: crescente5.c

```
1  #include <stdio.h>
2  void crescente5facil(float *x1, float *x2, float *x3, float *x4, float *x5) {
3     crescente4facil(x1,x2,x3,x4);
4     crescente4facil(x2,x3,x4,x5);
5     crescente4facil(x1,x2,x3,x4);
6  }
7  int main(){
8     float x1, x2, x3, x4, x5;
9     printf("n1: ");
10    scanf("%f", &x1);
11    printf("n2: ");
12    scanf("%f", &x2);
13    printf("n3: ");
14    scanf("%f", &x3);
15    printf("n4: ");
16    scanf("%f", &x4);
17    printf("n5: ");
18    scanf("%f", &x5);
19    crescente5facil(&x1,&x2,&x3,&x4,&x5);
20    printf("%f %f %f %f %f\n", x1, x2, x3, x4, x5);
21    return 0;
22 }
```

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 par. 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 algoritmo 28.

Algoritmo 28: 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 e 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 algoritmo 29. Este programa lê um número inteiro pelo terminal e o armazena na variável `n` (linhas 3, 4 e 5), assim como no 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 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.

Algoritmo 29: 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 }
```

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

Algoritmo 30: 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 algoritmo 30 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 algoritmo 30 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 algoritmo 31.

O comando `else` é muito útil para casos como o apresentado no algoritmo 31, 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.

Algoritmo 31: 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 algoritmo 32.

Algoritmo 32: 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`.

3.4 Subprogramas com condicionais

Condicionais também podem ser usados em subprogramas. Exemplo: Implemente um procedimento que receba um valor e diga se ele é negativo.

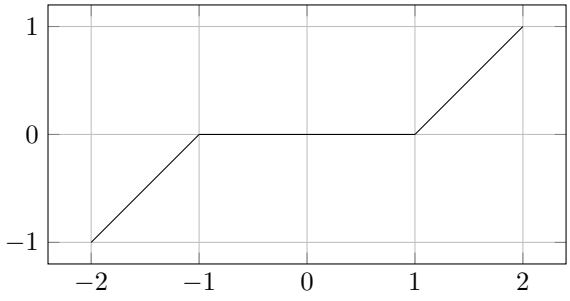
Algoritmo 33: negativo.c

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

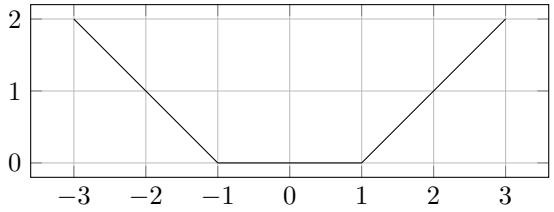
Exercícios

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

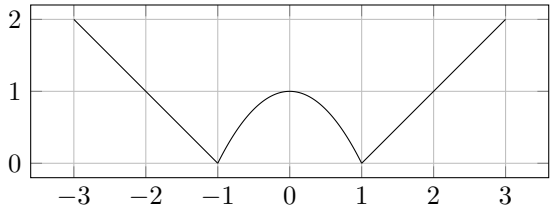
1. Implemente um procedimento que receba dois números e imprima o maior deles.
2. Implemente uma função que receba dois números e retorne o maior deles.
3. Implemente um procedimento que receba dois números e os imprima em ordem crescente.
4. Implemente um procedimento que receba dois ponteiros e coloque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



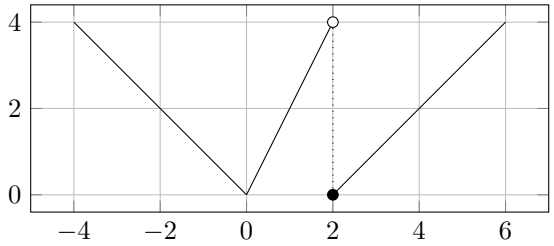
6. Implemente a função apresentada na figura a seguir.



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 algoritmo 34.

Algoritmo 34: 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 algoritmo 34 ser bem eficiente, o algoritmo 35 é bem mais intuitivo e legível.

Algoritmo 35: 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>=a){
4       return b;
5   }
6   if(b<=a && a<=c || b>=a && a>=c){
7       return a;
8   }
9   return c;
10 }
```

O algoritmo 35 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, u 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 $<$ 200	1.20
200 $<$ kWh $<$ 400	1.50
400 $<$ kWh $<$ 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 orientado a menu que calcule 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 um auto-chamada. Veja o algoritmo 36, onde o procedimento `contagem_regressiva` faz um auto-chamada na linha 5, reduzindo o valor do parâmetro em 1.

Algoritmo 36: 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.

Ordem dos comandos

Observe o algoritmo 37. É praticamente igual ao algoritmo 36, 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 do problema

Algoritmo 37: 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 }
```

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.

Funções intrinsecamente recursivas

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 algoritmo 38.

Algoritmo 38: fatorial.c

```
1  #include <stdio.h>
2  int fatorial(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 }
```

Várias funções matemáticas são definidas por recursão.

Exercícios

1. Implemente uma função recursiva que calcule a soma dos n primeiros números naturais.
2. Implemente um procedimento recursivo que imprima os n primeiros termos da sequência de Fibonacci.
3. Implemente uma função recursiva que retorne a quantidade de algarismos de um número inteiro.
4. Implemente uma função recursiva que calcule a soma dos algarismos de um número inteiro.
5. Implemente uma função recursiva que eleve um número inteiro a uma potência inteira.
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.
9. 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.

10. A quantidade de permutações dos algarismos de um número inteiro é função do fatorial da quantidade de casas decimais deste número. Podemos construir um identificador para cada permutação. Sendo assim, número com 2 casas decimais, teria 2 (2!) permutações com seus respectivos identificadores [Permutações:(ab, ba) Identificadores:(0, 1)]. Seguindo esta ideia, um número com 3 casas decimais pode ter 3!=6 permutações (abc, acb, bac, bca, cba, cab) com seus respectivos identificadores (0, 1, 2, 3, 4, 5). Faça uma função que a permutação de um número a partir do identificador desta permutação. Exemplo: permutação(123, 4)==321.

3.7 Laços de repetição

A maior parte dos algoritmos recursivos podem ser implementados de forma iterativa. Veja o exemplo do algoritmo 1. 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.

Algoritmo 39: 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 algoritmos recursivos têm sua versão iterativa tão parecida. Veja a comparação da contagem progressiva apresentada no algoritmo 1. Ainda assim, os dois algoritmos possuem valor inicial, condição para execução e diminuição do tamanho do problema. Porém, algoritmos recursivos sempre devem resolver parte do problema antes de diminuir o tamanho do problema, diferentemente de algoritmos recursivos.

Algoritmo 40: 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 algoritmos intrinsecamente recursivos podem ter versões iterativas com algumas adaptações, como mostra o algoritmo 1. Os dois algoritmos possuem a mesma condição de execução e a mesma redução de problema, porém a forma de guardar cada passo é diferente. O algoritmo recursivo delega a resolução de um problema menor para outra chamada da função antes de retornar seu resultado. O algoritmo iterativo, resolve a parte de cada iteração antes de enviar o problema menor para iteração seguinte. Por isso este usa uma variável auxiliar `f`.

Algoritmo 41: 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 }
```

Exercícios

1. Implemente um procedimento que imprima os n primeiros números naturais.
2. Implemente uma função que retorne a soma dos n primeiros números naturais.
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 um procedimento que imprima um triângulo de Pascal, com o padrão a seguir para $n = 4$.

```
1 1 1 1 1
1 2 3 4
1 3 6
1 4
1
```

7. Implemente uma função que calcule a soma $s(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$.
8. Implemente uma função que calcule a série $s(x, n) = 1 - \frac{x^2}{2} + \frac{x^3}{3} - \ldots \pm \frac{x^n}{n}$.
9. 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.
10. Implemente uma função que dados dois números inteiros, calcula a quantidade de números perfeitos que existem entre eles.
11. Implemente uma função que calcule quantos números primos existem entre dois números naturais.
12. Implemente um procedimento que imprima um número inteiro invertido ($12345 \Rightarrow 54321$).
13. Implemente uma função que verifique se um número é palíndromo ou não.
14. Implemente uma função que verifique se um número inteiro pode ser expresso pela soma de dois números primos.
15. Implemente uma procedimento que verifique se um número inteiro pode ser expresso pela multiplicação de dois números primos e imprima estes números quando possível.
16. 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.

Capítulo 4

Estruturas de dados homogêneas

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.

Exemplo: 32 alunos de uma determinada disciplina possuem notas que variam entre 0 e 10. Poderíamos armazenar as notas desses alunos em um vetor de 32 posições:

```
int quant_alunos;
float notas[32];
quant_alunos = 32;
```

Uma função para retornar a média da turma poderia ser:

Algoritmo 42: 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));
```

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 da função malloc e free.

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.

Algoritmo 43: 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 }
```

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 44: 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 }
```

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 32 alunos, só que agora, cada aluno terá 3 (três) notas. Felizmente a linguagem C não nos obriga a utilizar três vetores de 32 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 32x3. Desta forma, podemos armazenar, de forma legível, as três notas de cada aluno em uma única estrutura. Segue o exemplo abaixo:

Esta representação nos permite armazenar valores em linhas e colunas de forma clara. O código da quarta linha do programa acima (*float notas[32][3];*) declara uma matriz de 32 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[31][2]);
```

Algoritmo 45: Matrizes estáticas

```
1  int main() {
2    int quant_alunos, quant_notas;
3    float notas[32][3];
4    quant_alunos = 32;
5    quant_notas = 3;
6    printf("Preenchimento das notas:\n");
7    preenche_mat_float(notas, quant_alunos, quant_notas);
8    printf("Impressao das notas:\n");
9    print_mat_float(notas, quant_alunos, quant_notas);
10   printf("A media da turma e %f\n", media_mat_float(notas, quant_alunos,
       quant_notas));
11   return 0;
12 }
```

4.3 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 preenche_mat_float, printf_mat_float e media_mat_float.

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 da seguinte forma:

Algoritmo 46: 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 }
```

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.

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);
}
```

Agora a função media_mat_float pode ser implementada da seguinte forma:

Algoritmo 47: Calcula média de matrz 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 }
```

Assim a função media_mat_float ficou muito mais genérica, podendo agora, calcular a média para qualquer quantidade de alunos e notas.

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

Algoritmo 48: 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:

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 da seguinte forma:

Mesmo depois de liberada, a matriz notas recebeu NULL. Isso sempre deve ser feito porque a função libera_mat_float não apaga o valor contido em notas. Isto é, as posições de memória são apagadas, mas o endereço

Algoritmo 49: 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  }
```

Algoritmo 50: Calcula média da turma

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

de memória contido em notas continuará armazenado, mesmo depois de liberado. Esta vulnerabilidade pode ser explorada por hackers, a não ser que se atribua NULL ao ponteiro liberado.

As funções que não foram implementadas aqui devem ser implementadas como exercício.

Exercícios

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

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

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

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

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

5. 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);`

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

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

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

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

10. Faça um programa capaz de resolver problemas lineares com a mesma quantidade de variáveis e equações, sempre que possível.

4.4 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.

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.

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.

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.

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.

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.

8. Faça um procedimento que receba duas strings como parâmetros e copie o valor da 2ª para a 1ª.

9. Faça um procedimento que receba duas strings como parâmetros e concatene a 2ª à 1ª.

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". Repare 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);
```

4.5 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.

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.

Algoritmo 51: 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 }
```

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 52: 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:

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:

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:

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

Algoritmo 53: 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  }
```

Algoritmo 54: 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 }
```

Algoritmo 55: 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  }
```

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

4.6 Parâmetros da função principal

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

Algoritmo 56: 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 57: Imprime argumentos inteiros

```
1  int main(int argc, char** argv)
2  {
3   int i;
4   printf("%d\n", argc);
5   for(i=0; i<argc; i++)
6   {
7    printf("%d\n", atoi(argv[i]));
8   }
9   return 0;
10 }
```

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 ver uma forma de se 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:

O resultado deste programa seria:

```
Nome:  Joao de Deus
```

Algoritmo 58: Exemplo com registro

```
1  int main()
2  {
3    struct funcionario func;
4    float desc_perc;
5    desc_perc = 0.10f;
6    strcpy(func.nome, "Joao de Deus");
7    func.sal_contr = 500;
8    func.vant = 80;
9    func.desc = func.sal_contr * desc_perc;
10   func.sal_liq = func.sal_contr + func.vant - func.desc;
11   printf("Nome: %s\nSalario Contratural: %.2f\n Vantagens: %.2f\n\
12   Descontos: %.2f\n\
13   Salario Liquido: %.2f\n",
14   func.nome, func.sal_contr,
15   func.vant, func.desc, func.sal_liq);
16   return 0;
17 }
```

```
Salario Contratual:  500.00
Vantagens:  80.00
Descontos:  50.00
Salario Liquido:  530.00
```

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 59: Vetores de registros

```
1  int main()
2  {
3    struct funcionario func[TAM];
4    float desc_perc;
5    int quant_func = TAM;
6    printf("Desconto (%%): ");
7    scanf("%f", &desc_perc);
8    ler_dados_reg_func(func, quant_func);
9    calcular_folha_reg_func(func, quant_func, desc_perc);
10   imprimir_folha_reg_func(func, quant_func);
11   return 0;
12 }
```

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.

Algoritmo 60: 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 61: 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 62: 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 63: 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 }
```

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.

Algoritmo 64: 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 }
```

De forma análoga podemos definir um tipo Funcionario:

```
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 65: 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 66: 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 67: 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 68.

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

Algoritmo 68: 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  /* Coloca o valor das raizes nas variaveis x1 e x2, se existirem.
7  ** Retorna a quantidade de raizes encontradas. */
8  int baskara(float a, float b, float c, float *x1, float *x2) {
9    float d = delta(a, b, c);
10   /* Se o delta negativo a equacao nao possui raizes */
11   if(d < 0)
12     return 0;
13   /* Se o delta for zero so existe uma raiz */
14   if(d == 0){
15     *x1 = *x2 = ((-1)*b)/(2*a);
16     return 1;
17   }
18   /* Se o delta for positivo existem duas raizes reais */
19   *x1 = ((-1)*b+sqrt(d))/(2*a);
20   *x2 = ((-1)*b-sqrt(d))/(2*a);
21   return 2;
22 }
```

permite que utilizemos, dentro do arquivo .c, uma função declarada no arquivo .h, acima de 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 69.

Algoritmo 69: 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.