Programação em C

Francisco de Assis Boldt

27 de abril 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.

Algoritmo 3: soma5e7.c

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

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 a linha 3. 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 int main() {
3    int x, y;
4    x = 5;
5    y = 7;
6    printf("%d + %d = %d\n", x, y, x+y);
7    return 0;
8 }
```

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.

Algoritmo 5: hello_sub.c

```
1 #include <stdio.h>
2 void hello(){
3   printf("Hello World!\n");
4 }
5 int main() {
6   hello();
7   return 0;
8 }
```

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.

2.2.1 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`.

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(8, 9);
8     imprime_soma(2, 3);
9     return 0;
10 }
```

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

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.

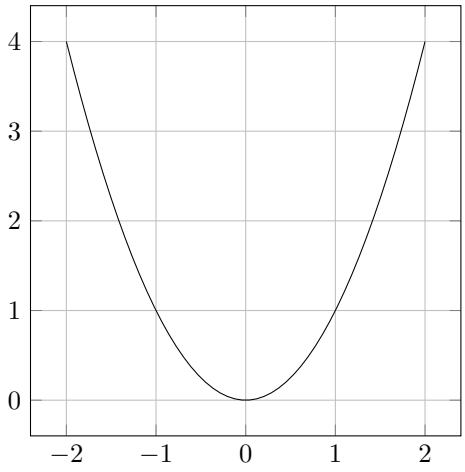

Figura 2.1: $f(x) = x^2$

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.

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    float x = -2;
7    printf("f(%f) = %f", x, quadrado(x));
8    x = 2;
9    printf("f(%f) = %f", x, quadrado(x));
10   return 0;
11 }
```

2.3.1 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    float x = -2;
5    printf("f(%f) = %f\n", x, cos(x));
6    x = 0;
7    printf("f(%f) = %f\n", x, cos(x));
8    x = 1;
9    printf("f(%f) = %f\n", x, cos(x));
10   x = 3.1415;
11   printf("f(%f) = %f\n", x, cos(x));
12   return 0;
13 }
```

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.

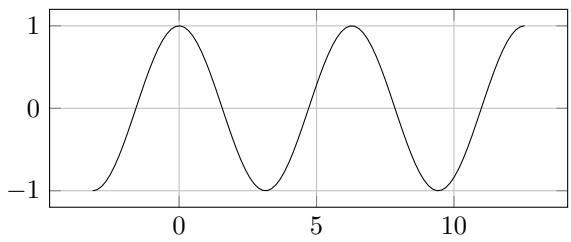

Figura 2.2: $f(x) = cos(x)$

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

2.3.2 Exercícios com funções contínuas

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.

2.4.1 Função rampa

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

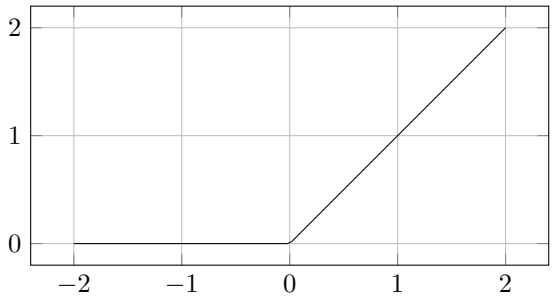

Figura 2.3: $f(x) = rampa(x)$

[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.

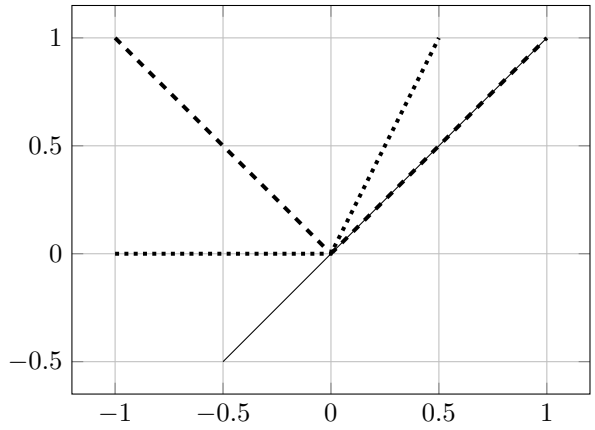

Figura 2.4: $f_1(x) = x$, $f_2(x) = |x|$, $f_3(x) = f_1(x) + f_2(x)$.

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`.

2.4.2 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 $f(x) = \frac{x-|x|}{2}$.

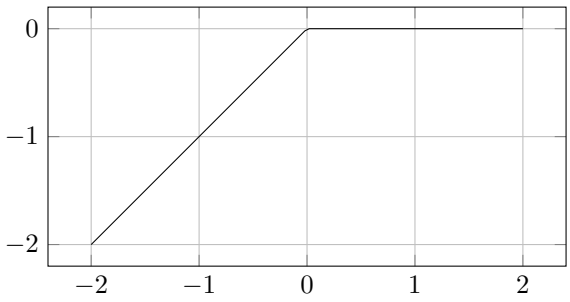

Figura 2.5: $f(x) = rampainv(x)$

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 }
```

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

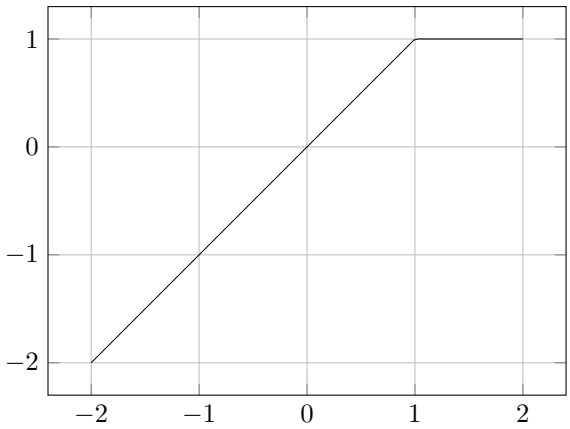

Figura 2.6: $f(x) = rampainvmax1(x)$

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

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",- 2.0, rampainvmax1min0(-2));
11   printf("f(%f) = %f\n", -1.0, rampainvmax1min0(-1));
12   printf("f(%f) = %f\n", 0.0, rampainvmax1min0(0));
13   printf("f(%f) = %f\n", 1.0, rampainvmax1min0(1));
14   printf("f(%f) = %f\n", 2.0, rampainvmax1min0(2));
15   return 0;
16 }
```

2.4.4 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.

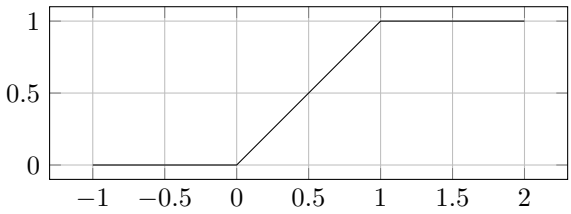

Figura 2.7: $f(x) = rampainvmax1min0(x)$

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",- 2.0, rampainvmax1min0(-2));
14   printf("f(%f) = %f\n", -1.0, rampainvmax1min0(-1));
15   printf("f(%f) = %f\n", 0.0, rampainvmax1min0(0));
16   printf("f(%f) = %f\n", 1.0, rampainvmax1min0(1));
17   printf("f(%f) = %f\n", 2.0, rampainvmax1min0(2));
18   return 0;
19 }
```

2.4.5 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.

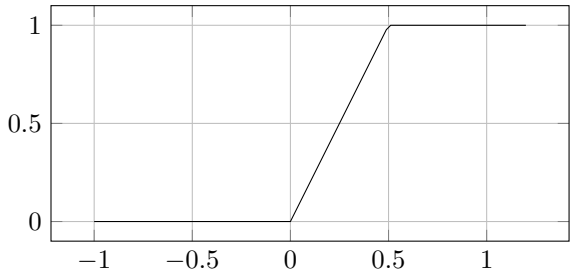

Figura 2.8: $f(x) = rampaingreme(x)$

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.

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

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", -0.5, rampaingreme(-0.5));
16   printf("f(%f) = %f\n", -0.25, rampaingreme(-0.25));
17   printf("f(%f) = %f\n", 0.0, rampaingreme(0));
18   printf("f(%f) = %f\n", 0.25, rampaingreme(0.25));
19   printf("f(%f) = %f\n", 0.5, rampaingreme(0.5));
20   printf("f(%f) = %f\n", 1.0, rampaingreme(1));
21   printf("f(%f) = %f\n", 1.5, rampaingreme(1.5));
22   return 0;
23 }
```

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. 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.

2.4.6 Função degrau

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

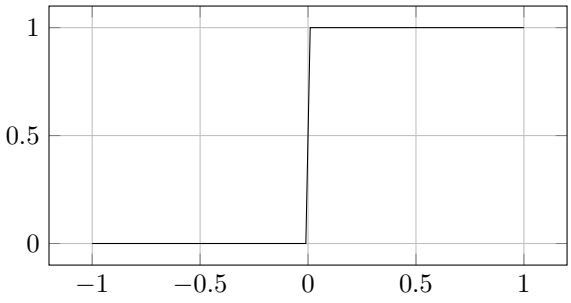

Figura 2.9: $f(x) = degrau(x)$

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

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

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 exercícios a seguir mostram algumas aplicações dela.

2.4.7 Exercícios usando a função degrau

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 2.4.7 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 }
```

Note que a função degrau ficou tão simples ao ponto de 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. Observe que a implementação da função `maior2` no algoritmo 17 pode ser feita apenas substituindo a função `maior` pelo operador >.

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", 5.0, 6.0, maior2(5,6));
8    printf("%f > %f ==> %f\n", 5.0, 5.0, maior2(5,5));
9    return 0;
10 }
```

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 }
```

A implementação do procedimento `crescente2` utilizando operadores relacionais é bem mais simples do que usando a função degrau. Porém, essa implementação ainda está complexa para resolver um problema simples. Para entender melhor, tente implementar, usando apenas operadores aritméticos e relacionais, um procedimento que receba três números reais e os imprima em ordem crescente.

Algoritmo 19: crescente3.c

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

O 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 números.

2.6 Passagem de parâmetros por referência

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`. Então, 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.

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.

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

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 }
```

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`.

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

Algoritmo 27: crescente5.c

```
1  void crescente5facil(float *x1, float *x2, float *x3, float *x4, float *x5) {
2    crescente4facil(x1,x2,x3,x4);
3    crescente4facil(x2,x3,x4,x5);
4    crescente4facil(x1,x2,x3,x4);
5  }
6  int main(){
7    float x1, x2, x3, x4, x5;
8    int i=1;
9    printf("n%d: ",i++);
10   scanf("%f", &x1);
11   printf("n%d: ",i++);
12   scanf("%f", &x2);
13   printf("n%d: ",i++);
14   scanf("%f", &x3);
15   printf("n%d: ",i++);
16   scanf("%f", &x4);
17   printf("n%d: ",i++);
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 }
```

Exercícios

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

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

2. 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.

3. 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.

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

Algoritmo 28: celsius2fahrenheit.c

```
1  #include <stdio.h>
2  float celsius2fahrenheit(float c) {
3    return c*9/5+32;
4  }
5  int main() {
6    float c;
7    printf("Celsius: ");
8    scanf("%f", &c);
9    printf("Fahrenheit: %f\n", celsius2fahrenheit(c));
10   return 0;
11 }
```

Algoritmo 29: retangulo.c

```
1  #include <stdio.h>
2  float area_retangulo(float base, float altura) {
3    return base*altura;
4  }
5  float perimetro_retangulo(float base, float altura) {
6    return 2*base + 2*altura;
7  }
8  int main() {
9    float base, altura;
10   printf("base: ");
11   scanf("%f", &base);
12   printf("altura: ");
13   scanf("%f", &altura);
14   printf("area: %f\nperimetro: %f\n",
15          area_retangulo(base, altura),
16          perimetro_retangulo(base, altura));
17   return 0;
18 }
```

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 30.

Algoritmo 30: 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 e 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 31. 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 31: 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 e 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 32: 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 e impar.\n", n);
8   }
9   if(n%2 == 0){
10    printf("%d e par.\n", n);
11  }
12 }
```

O algoritmo 32 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 32 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 33.

O comando `else` é muito útil para casos como o apresentado no algoritmo 33, 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 33: 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 e impar.\n", n);
8    } else {
9      printf("%d e par.\n", n);
10   }
11 }
```

3.3 Exercícios

1. Implemente um procedimento que receba um valor e diga se ele é negativo.
2. Implemente um procedimento, usando os comandos `if` e `else`, que receba dois números e imprima o maior deles.
3. Implemente uma função, usando os comandos `if` e `else`, que receba dois números e retorne o maior deles.
4. Implemente um procedimento, usando o comando `if`, que receba dois números e os imprima em ordem crescente.
5. Implemente um procedimento, usando o comando `if`, que receba dois ponteiros e coloque seus valores em ordem crescente.

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 34: 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 35: 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 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 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`[1]

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 extenção '.h', que significa Header, ou cabeçalho.

Algoritmo 36: 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 raizes da equação, como mostra o algoritmo 37.

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

[1] *-lm* é necessário devido ao uso da biblioteca *math.h*.

Algoritmo 37: 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 38.

Algoritmo 38: 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.