10.2 Exercícios

Exercício 10.1 (KING, 2008): Escreva um programa que leia duas datas e que as apresente em ordem crescente. Cada data deve ser armazenada usando o tipo Data que contém três membros inteiros: dia, mês e ano. A comparação entre as datas deve ser feita utilizando a função `compararData`. Se `d1` for menor que `d2`, um valor negativo deve ser retornado. Se `d1` for maior que `d2`, um valor positivo deve ser retornado. Se `d1` e `d2` forem a mesma data, 0 deve ser retornado. A impressão das datas deve ser feita utilizando a função `imprimirData`. As definições das funções são:

- `int compararData( const Data *d1, const Data *d2 )`
- `void imprimirData( const Data *data )`

Arquivo com a solução: ex10.1.c

Entrada

```
Data 1
    dia: 1
    mes: 2
    ano: 1990
Data 2
    dia: 1
    mes: 2
    ano: 2000
```

Saída

```
01/02/1990 <= 01/02/2000
```

Entrada

```
Data 1
    dia: 20
    mes: 3
    ano: 2000
Data 2
    dia: 31
    mes: 2
    ano: 2000
```

Saída

```
31/02/2000 <= 20/03/2000
```

Entrada

```
Data 1
    dia: 25
    mes: 2
    ano: 2019
Data 2
    dia: 24
    mes: 2
    ano: 2019
```

Saída

```
24/02/2019 <= 25/02/2019
```

Entrada

```
Data 1
    dia: 30
    mes: 1
    ano: 2000
Data 2
    dia: 30
    mes: 1
    ano: 2000
```

Saída

```
30/01/2000 <= 30/01/2000
```

Exercício 10.2 (KING, 2008): Escreva um programa que leia uma data e que a partir da mesma obtenha o dia do ano correspondente. Cada data deve ser armazenada usando o tipo Data que contém três membros inteiros: dia, mês e ano. O cálculo do dia do ano deve ser feito utilizando a função `diaDoAno`. A definição das função é:

- `int diaDoAno( const Data *data )`

Arquivo com a solução: ex10.2.c

Entrada

```
dia: 1
mes: 4
ano: 2014
```

Saída

```
O dia do ano da data 01/04/2014 eh 91.
```

Entrada

```
dia: 1
mes: 4
ano: 2016
```

Saída

```
O dia do ano da data 01/04/2016 eh 92.
```

Exercício 10.3 (KING, 2008): Escreva um programa que leia uma quantidade de segundos e que, a partir desse valor, gere a quantidade de horas, minutos e segundos correspondentes, gerando como resultado uma instância do tipo Hora, que contém três membros inteiros: hora, minuto e segundo. Esse cálculo deve ser feito através da função `gerarHora`. A impressão da hora deve ser feita utilizando a função `imprimirHora`. As definições das funções são:

- `Hora gerarHora( int quantidadeSegundos )`
- `void imprimirHora( const Hora *hora )`

Arquivo com a solução: ex10.3.c

Entrada

```
Segundos: 254783
```

Saída

```
Hora correspondente: 70:46:23
```

Exercício 10.4 (KING, 2008): Escreva um programa que leia dois números complexos, representados pelo tipo Complexo, e que calcule a soma dos mesmos. O tipo Complexo contém dois membros decimais: real e imaginário. A soma dos números complexos

deve ser feita através da função `somar` que recebe dois números complexos e que retorna um novo número complexo que representa a soma dos números passados. A impressão do valor resultante deve ser feita através da função `imprimirComplexo`. As definições das funções são:

- `Complexo somar( const Complexo *c1, const Complexo *c2 )`
- `void imprimirComplexo( const Complexo *c )`

Arquivo com a solução: ex10.4.c

Entrada

```
Complexo 1
    Parte real: 10
    Parte imaginaria: 5
Complexo 2
    Parte real: 3
    Parte imaginaria: 9
```

Saída

```
(10.00 + 5.00i) + (3.00 + 9.00i) = (13.00 + 14.00i)
```

Exercício 10.5 (KING, 2008): Escreva um programa que leia duas frações, representados pelo tipo Fracao, e que calcule sua soma, subtração, multiplicação e divisão. O tipo Fracao contém dois membros decimais: numerador e denominador. As operações com as frações devem ser feitas através das funções `somar`, `subtrair`, `multiplicar` e `dividir`. A soma e a subtração de frações envolve a obtenção do mínimo múltiplo comum, entretanto, para simplificar a implementação, caso os denominadores sejam diferentes, calcule o denominador comum como a multiplicação dos dois denominadores. Além disso, as frações não precisam ser simplificadas. Essas quatro funções recebem duas frações e retornam uma nova fração com o resultado esperado. A impressão das frações resultantes deve ser feita através da função `imprimirFracao`. As definições das funções são:

- `Fracao somar( const Fracao *f1, const Fracao *f2 )`
- `Fracao subtrair( const Fracao *f1, const Fracao *f2 )`
- `Fracao multiplicar( const Fracao *f1, const Fracao *f2 )`
- `Fracao dividir( const Fracao *f1, const Fracao *f2 )`
- `void imprimirFracao( const Fracao *f )`

Arquivo com a solução: ex10.5.c

Entrada

```
Fracao 1
    Numerador: 1
    Denominador: 2
Fracao 2
    Numerador: 2
    Denominador: 3
```

Saída

```
1.00/2.00 + 2.00/3.00 = 7.00/6.00
1.00/2.00 - 2.00/3.00 = -1.00/6.00
1.00/2.00 * 2.00/3.00 = 2.00/6.00
1.00/2.00 / 2.00/3.00 = 3.00/4.00
```

Exercício 10.6 (KING, 2008): Escreva um programa que leia os componentes vermelho, verde e azul que são usados para representar uma cor e que crie uma instância do tipo Cor através da função `novaCor`. Essa função deve validar a entrada, ou seja, cada um dos componentes da cor deve estar, obrigatoriamente, no intervalo de 0 a 255 inclusive. Caso um valor menor que zero seja fornecido, o valor zero deve ser atribuído ao componente respectivo. Caso um valor maior de 255 seja fornecido, o valor 255 será atribuído. Utilize a função `imprimirCor` para imprimir os dados da cor. As definições das funções são:

- `Cor novaCor( int vermelho, int verde, int azul )`
- `void imprimirCor( const Cor *c )`

Arquivo com a solução: ex10.6.c

Entrada

```
Vermelho: 50
Verde: 60
Azul: 70
```

Saída

```
Cor: rgb( 50, 60, 70 )
```

Entrada

```
Vermelho: -10
Verde: -10
Azul: -10
```

Saída

```
Cor: rgb( 0, 0, 0 )
```

Entrada

```
Vermelho: 260
Verde: 180
Azul: 280
```

Saída

```
Cor: rgb( 255, 180, 255 )
```

Exercício 10.7 (KING, 2008): Com base no exercício anterior, escreva um programa que leia uma cor e que apresente os valores dos seus componentes utilizando as funções `getVermelho`, `getVerde` e `getAzul` que retornam, respectivamente, os componentes vermelho, verde e azul de uma cor. Utilize a função `novaCor` para criar a Cor base. As definições das funções são:

- `int getVermelho( const Cor *c )`
- `int getVerde( const Cor *c )`
- `int getAzul( const Cor *c )`

Arquivo com a solução: ex10.7.c

Entrada

```
Vermelho: 50
Verde: 60
Azul: 70
```

Saída

```
Cor: rgb( 50, 60, 70 )
getVermelho(): 50
getVerde(): 60
getAzul(): 70
```

Entrada

```
Vermelho: -10
Verde: -10
Azul: -10
```

Saída

```
Cor: rgb( 0, 0, 0 )
getVermelho(): 0
getVerde(): 0
getAzul(): 0
```

Entrada

```
Vermelho: 260
Verde: 180
Azul: 280
```

Saída

```
Cor: rgb( 255, 180, 255 )
getVermelho(): 255
getVerde(): 180
getAzul(): 255
```

Exercício 10.8 (KING, 2008): Com base no exercício anterior, escreva um programa que leia uma cor e reconfigure os seus membros utilizando as funções `setVermelho`, `setVerde` e `setAzul`. As mesmas validações da função `novaCor`, já implementada, se aplicam. Utilize a função `novaCor` para criar a Cor base. As definições das funções são:

- `void setVermelho( Cor *c, int vermelho )`
- `void setVerde( Cor *c, int verde )`

- `void setAzul( Cor *c, int azul )`

Arquivo com a solução: ex10.8.c

Entrada

```
Vermelho: 50
Verde: 60
Azul: 70
Novo vermelho: -1
Novo verde: 600
Novo azul: 190
```

Saída

```
Cor: rgb( 50, 60, 70 )
Cor alterada: rgb( 0, 255, 190 )
```

Exercício 10.9 (KING, 2008): Com base no exercício anterior, escreva um programa que leia uma cor e que gere uma versão mais escura dessa cor utilizando a função `escurecer`. A função `escurecer` deve multiplicar cada membro da cor por 0.7, truncando a parte decimal no resultado para a nova cor. Utilize a função `novaCor` para criar a Cor base. A definição da função é:

- `Cor escurecer( const Cor *c )`

Arquivo com a solução: ex10.9.c

Entrada

```
Vermelho: 255
Verde: 180
Azul: 90
```

Saída

```
Cor base: rgb( 255, 180, 90 )
Cor escurecida: rgb( 178, 125, 62 )
```

Exercício 10.10 (KING, 2008): Com base no exercício anterior, escreva um programa que leia uma cor e que gere uma versão mais clara dessa cor utilizando a função `clarear`. A função `clarear` deve dividir cada membro por 0.7, truncando a parte decimal no resultado para a nova cor. Entretanto, existem alguns casos especiais que

devem ser observados: 1) Se o valor de todos os componentes da cor original forem iguais a zero, a nova cor deve ser gerada com todos os componentes valendo 3; 2) Caso algum componente da cor original for maior que 0, mas menor que 3, deve-se configurá-lo como 3 na nova cor antes da divisão por 0,7; 3) Se a divisão por 0,7 de um membro da cor original resultar em algum valor maior que 255, deve-se configurar, na nova cor, esse componente com o valor 255. Utilize a função `novaCor` para criar a Cor base. A definição da função é:

- `Cor clarear( const Cor *c )`

Arquivo com a solução: ex10.10.c

Entrada

```
Vermelho: 10
Verde: 30
Azul: 40
```

Saída

```
Cor base: rgb( 10, 30, 40 )
Cor clareada: rgb( 14, 42, 57 )
```

Entrada

```
Vermelho: 0
Verde: 0
Azul: 0
```

Saída

```
Cor base: rgb( 0, 0, 0 )
Cor clareada: rgb( 3, 3, 3 )
```

Entrada

```
Vermelho: 1
Verde: 2
Azul: 1
```

Saída

```
Cor base: rgb( 1, 2, 1 )
Cor clareada: rgb( 4, 4, 4 )
```

Entrada

```
Vermelho: 100
Verde: 150
Azul: 230
```

Saída

```
Cor base: rgb( 100, 150, 230 )
Cor clareada: rgb( 142, 214, 255 )
```

Exercício 10.11 (KING, 2008): Escreva um programa que leia as coordenadas de dois pontos, armazenadas no tipo Ponto, que contém os membros inteiros: x e y. A partir dos dois pontos lidos, uma instância do tipo Retangulo, que contém dois membros do tipo Ponto (`superiorEsquerdo` e `inferiorDireito`) deve ser criada, usando a função `novoRetangulo`, e sua área deve ser calculada pela função `calcularArea`. Para imprimir o Retangulo, utilize a função `imprimirRetangulo`. Considere que o sistema de coordenadas adotado é o mesmo de um plano cartesiano tradicional. A apresentação de valores inteiros com sinal deve ser feita usando a opção + no especificador de formato %d. Por exemplo, %+02d formata um inteiro, com no mínimo duas casas, preenchendo com zeros se necessário, além de exibir o sinal. As definições das funções são:

- `Retangulo novoRetangulo( const Ponto *sEsq, const Ponto *iDir )`
- `int calcularArea( const Retangulo *r )`
- `void imprimirRetangulo( const Retangulo *r )`

Arquivo com a solução: ex10.11.c

Entrada

```
Ponto superior esquerdo
    x: 10
    y: 40
Ponto inferior direito
    x: 60
    y: 10
```

Saída

```
(+10, +40) =====|
|               |
|               |
|===== (+60, +10)
Area: 1500
```

Entrada

```
Ponto superior esquerdo
    x: -60
    y: -10
Ponto inferior direito
    x: -10
    y: -50
```

Saída

```
(-60, -10) =====|
|               |
|               |
|===== (-10, -50)
Area: 2000
```

Entrada

```
Ponto superior esquerdo
    x: -60
    y: 30
Ponto inferior direito
    x: 60
    y: -30
```

Saída

```
(-60, +30) =====|
|               |
|               |
|===== (+60, -30)
Area: 7200
```

Exercício 10.12 (KING, 2008): Com base no exercício anterior, escreva um programa que crie uma instância do tipo Retangulo e que calcule seu ponto central, utilizando, para isso, a função `obterCentro`. Considere que o sistema de coordenadas adotado é o mesmo de um plano cartesiano tradicional. A definição da função é:

- `Ponto obterCentro( const Retangulo *r )`

Arquivo com a solução: ex10.12.c

Entrada

```
Ponto superior esquerdo
    x: 10
    y: 40
Ponto inferior direito
    x: 60
    y: 10
```

Saída

```
(+10, +40) =====|
|               |
|               |
|===== (+60, +10)
Centro: (+35, +25)
```

Entrada

```
Ponto superior esquerdo
    x: -60
    y: -10
Ponto inferior direito
    x: -10
    y: -50
```

Saída

```
(-60, -10) =====|
|               |
|               |
|===== (-10, -50)
Centro: (-35, -30)
```

Entrada

```
Ponto superior esquerdo
    x: -60
    y: 30
Ponto inferior direito
    x: 60
    y: -30
```

Saída

```
(-60, +30) =====|
|               |
|               |
|===== (+60, -30)
Centro: (+0, +0)
```

Exercício 10.13 (KING, 2008): Com base no exercício anterior, escreva um programa que crie uma instância do tipo Retangulo e a mova em uma quantidade arbitrária de unidades em x e em y, utilizando, para isso, a função `mover`. Considere que o sistema de coordenadas adotado é o mesmo de um plano cartesiano tradicional. A definição da função é:

- `void mover( Retangulo *r, int x, int y )`

Arquivo com a solução: ex10.13.c

Entrada

```
Ponto superior esquerdo
    x: 10
    y: 40
Ponto inferior direito
    x: 60
    y: 10
Mover em x: 50
Mover em y: -10
```

Saída

```
Retangulo original:
(+10, +40) =====|
|               |
|               |
|===== (+60, +10)
Retangulo movido:
(+60, +30) =====|
|               |
|               |
|===== (+110, +0)
```

Exercício 10.14 (KING, 2008): Com base no exercício anterior, escreva um programa que crie uma instância do tipo Retangulo e verifique se cinco pontos, também fornecidos pelo usuário, estão contidos ou não dentro desse retângulo. Para isso, utilize a função `contem`. Considere que o sistema de coordenadas adotado é o mesmo de um plano cartesiano tradicional. A definição da função é:

- `bool contem( const Retangulo *r, const Ponto *p )`

Arquivo com a solução: ex10.14.c

Entrada

```
Retangulo
Ponto superior esquerdo
    x: -60
    y: 30
Ponto inferior direito
    x: 60
    y: -30
Pontos
  Ponto 1
    x: -70
    y: 20
  Ponto 2
    x: -50
    y: 20
  Ponto 3
    x: 60
    y: 30
  Ponto 4
    x: 55
    y: -20
  Ponto 5
    x: 40
    y: -40
```

Saída

```
(-70, +20): nao contido!
(-50, +20): contido!
(+60, +30): contido!
(+55, -20): contido!
(+40, -40): nao contido!
```

Exercício 10.15: Com base no exercício anterior, escreva um programa que crie duas instâncias do tipo Retangulo e verifique se os dois retângulos representados por essas instâncias se interceptam. Para isso, utilize a função `intercepta`. Considere que o sistema de coordenadas adotado é o mesmo de um plano cartesiano tradicional. Dica: você pode utilizar a função contem do exercício anterior. A definição da função é:

- `bool intercepta( const Retangulo *r1, const Retangulo *r2 )`

Arquivo com a solução: ex10.15.c

Entrada

```
Retangulo 1
Ponto superior esquerdo
    x: 10
    y: 40
Ponto inferior direito
    x: 60
    y: 10
Retangulo 2
Ponto superior esquerdo
    x: 30
    y: 50
Ponto inferior direito
    x: 50
    y: 20
```

Saída

```
Os retangulos se interceptam!
```

Entrada

```
Retangulo 1
Ponto superior esquerdo
    x: 10
    y: 40
Ponto inferior direito
    x: 60
    y: 10
Retangulo 2
Ponto superior esquerdo
    x: -30
    y: 60
Ponto inferior direito
    x: 20
    y: -10
```

Saída

```
Os retangulos se interceptam!
```

Entrada

```
Retangulo 1
Ponto superior esquerdo
    x: 10
    y: 40
Ponto inferior direito
    x: 60
    y: 10
Retangulo 2
Ponto superior esquerdo
    x: 30
    y: 100
Ponto inferior direito
    x: 60
    y: 50
```

Saída

```
Os retangulos nao se interceptam!
```