Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Linguagem de Programação I • DIM0120

◁ Exercícios de Programação #1 ▷

22 de maio de 2021

Orientações gerais

Nos exercícios seguintes existem duas categorias de problemas a serem implementados em C++: aqueles que pedem para você elaborar um **programa** e aqueles que pedem para escrever uma **função**.

Para a primeira categoria você deve assumir que os dados de entrada para cada programa devem ser lidos da **entrada padrão**, `stdin`, através da operação de *extração* sobre o objeto `std::cin`. Similarmente, a saída de cada programa deve ser enviada para a **saída padrão**, `stdout`, através do operador de *inserção* sobre o objeto `std::cout`. Veja o exemplo abaixo:

```
int x; // Variável que armazenará um valor.
std::cin >> std::ws >>  x; // Lendo um valor da entrada padrão em 'x'.
// O objeto 'std::ws' ignora todos os espaços em branco que precedem
// o valor a ser lido do fluxo de entrada.
std::cout << x << '\n'; // Escrevendo valor na saída padrão.
```

Para as questões que solicitam a **implementação de uma função,** você deve seguir exatamente a assinatura da função indicada na questão, se for o caso.

É importante que você siga a risca as instruções sobre envio de informações para saída padrão e obedeça as assinaturas de funções indicadas, uma vez que suas respostas serão validadas de forma automática, através de testes já desenvolvidos.

1 Negativos 5

Descrição

Escreva um programa em C++ que recebe 5 valores inteiros da entrada padrão, conta quantos destes valores são negativos e imprime esta informação na saída padrão. Veja abaixo exemplo de entrada e saída esperada.

Formato Entrada/Saída

Exemplo de entrada 1:	Saída correspondente 1:
-1 -2 0 1 2	2

Exemplo de entrada 2:	Saída correspondente 2:
1 2 3 4 5	0

Conhecimentos necessários

Leitura de entrada padrão, escrita em saída padrão, laços, condicionais.

2 Soma Vizinhos

Descrição

Escreva um programa em C++ que lê um número não determinado de pares de valores inteiros, m e n, onde $-10000 \leq m, n \leq 1000$, e para cada par calcula e escreve a soma dos n primeiros inteiros **consecutivos** à partir de m (inclusive), se $n > 0$; ou a soma dos n primeiros inteiros **antecedentes** à partir de m (inclusive), se $n < 0$. Por exemplo, se a entrada for "$1 \quad 5$" o programa deve imprimir na saída padrão o resultado de $1 + 2 + 3 + 4 + 5$, ou seja, 15. Por outro lado, se a entrada for "$1 \quad -5$" o programa deve imprimir na saída padrão o resultado de $1 + 0 + (-1) + (-2) + (-3)$, ou seja, -5.

Formato Entrada/Saída

Exemplo de entrada 1:	Saída correspondente 1:
4 6	39

Exemplo de entrada 2:	Saída correspondente 2:
10 -4	34

Exemplo de entrada 3:	Saída correspondente 3:
39 0	39

Conhecimentos necessários

Leitura de entrada padrão, escrita em saída padrão, laços com incremento não convencional, condicionais.

3 Intervalos

Descrição

Escreva um programa em C++ que lê um número não conhecido de valores inteiros e conta quantos deles estão em cada um dos intervalos $[0, 25)$, $[25, 50)$, $[50, 75)$, $[75, 100)$ e fora desses intervalos. Para ler um número indeterminado de valores basta incluir o comando de extração associado ao `std::cin` como condição de parada em um laço (ver abaixo).

```
int x;
while( cin >> std::ws >> x ) {
    // Realização da contagem de ocorrências nos intervalos
    ...
}
// Exibir contagem para os intervalos solicitados.
```

Após encerrada a entrada de dados, o programa deve imprimir a **porcentagem** de números para cada um dos quatro intervalos e de números fora dos intervalos, nessa ordem, um por linha e representados com quatro casas de precisão. A precisão pode ser definida com `std::setprecision(4)`.

Formato Entrada/Saída

Exemplo de entrada 1:

```
-9 -8
1 5 15 20
25 30 35 40 45 46 47
50 55 60
78 85 90 99
100 120
```

Saída correspondente 1:

```
18.18
31.82
13.64
18.18
18.18
```

Exemplo de entrada 2:

```
25 30 35 40 45 46 47
78 85 90 99
100 120
```

Saída correspondente 2:

```
0
53.85
0
30.77
15.38
```

Conhecimentos necessários

Leitura de entrada padrão, escrita em saída padrão, laços, condicionais, *type casting*, saída formatada, uso de vetor. *Dicas*: o uso de estrutura de dados **tabela de dispersão** pode ajudar a criar um código mais eficiente e mais simples.

4 Fibonacci

Descrição

Implemente uma função em C++ chamada `fib_below_n()` que recebe um valor *inteiro positivo* `n` e armazena os termos da série de Fibonacci **inferiores** a `n` em um `std::vector` `v` e retorna esse objeto para o cliente via retorno de função. A classe `std::vector` representa um tipo de contêiner que faz parte da biblioteca padrão do STL[1] e representa a estrutura de dados **lista dinâmica**. Essa função deve ter a seguinte assinatura:

```
std::vector<int> fib_below_n( unsigned int n );
```

A sequência de Fibonacci define-se como tendo os dois primeiros termos iguais a 1 e cada termo seguinte é a soma dos dois termos imediatamente anteriores. Desta forma se fosse fornecido ao programa uma entrada `n` $= 15$ o mesmo deveria produzir a seguinte sequência de termos da série: $\{1, 1, 2, 3, 5, 8, 13\}$.

Conhecimentos necessários

Utilização de função, manipulação de classes do STL, retorno de função, laço, séries matemáticas.

[1] *Standard Template Library:* é um conjunto de funções e classes template do C++ que oferece contêineres (estruturas de dados), algoritmos, iteradores e funções.

5 Minmax

Descrição

Escreva uma função em C++ chamada `min_max()` que recebe como parâmetro um arranjo unidimensional (vetor) `V` com `n` números inteiros, identifica e retorna um *par* de valores correspondentes aos índices da **primeira** ocorrência do **menor** elemento e da **última** ocorrência do **maior** elemento presente em `V`. Por exemplo, se a entrada fosse `V={5,4,1,3,1,10,7,10,6,8}`, a função deveria retornar o par `{2,7}`, correspondente às posições do primeiro '1' em `V[2]` e do último '10' na posição `V[7]`.

Um par de valores pode ser retornado por uma função através de um `struct` ou utilizando a classe utilitária `std::pair`. Essa função deve ter a seguinte assinatura:

```
std::pair<size_t,size_t> min_max( int V[], size_t n );
```

Conhecimentos necessários

Utilização de função, passagem de vetor por parâmetro, utilização de classes do STL, retorno de função com tipo composto, condicionais, laços.

6 Inverter

Descrição

Escreva uma função em C++ chamada `reverse()` que recebe como parâmetro uma *referência* para um **arranjo estático unidimensional** de strings, implementado com `std::array`, e inverte a ordem dos seus elementos da forma mais eficiente possível. Por exemplo, considere o vetor `A` contém as seguintes strings: [*"um"*, *"dois"*, *"três"*, *"quatro"*], após a execução da função o vetor `A` deverá ter seus elementos na seguinte ordem: [*"quatro"*, *"três"*, *"dois"*, *"um"*].

A classe `std::array` representa um vetor estático de memória contígua que faz parte da biblioteca padrão do STL. A função `reverse` deve ter a seguinte assinatura:

```
template <size_t SIZE>
void reverse( std::array<std::string,SIZE> & arr );
```

Note nessa assinatura a presenta da indicação de *template*. A variável `SIZE` (inteiro longo sem sinal) será definida em *tempo de compilação* pelo cliente que invocar sua função. Uma das vantagens de usar uma classe para definir o vetor, ao invés de usar um vetor tradicional do C++, é que a classe possui várias funcionalidades já implementadas e disponíveis para uso. Por exemplo, se você desejar recuperar a quantidade de elementos em `arr` basta invocar o método `size()`, como em:

```
size_t tamanho = arr.size();
```

com isso, não precisamos passar o comprimento de `arr` como mais um argumento para a função, da mesma forma que fizemos na Questão 5. Outro ponto a destacar é o uso do operador de referência, `&`, associado a variável `arr`. Esse operador faz com que qualquer alteração realizada sobre `arr` tenha efeito permanente quando a função retornar sua execução para o cliente. Por esse motivo que a função não precisa retornar nenhum valor (i.e. `void`), pois a "comunicação" de informações com o cliente será feito através da **passagem de parâmetro por referência**.

Conhecimentos necessários

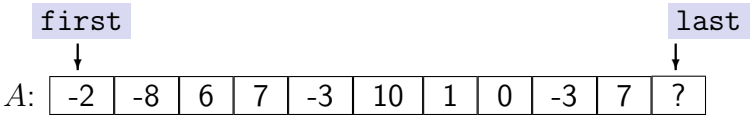
Utilização de função, cadeia de caracteres (string), passagem de parâmetro por referência, classe do STL, laços, manipulação da variável de controle de laço, operação *swap*.
Observação: Não é necessário criar um outro vetor para inverter os valores presentes no vetor passado por referência. A inversão pode ser feito *internamente*, ou seja, dentro do próprio vetor, com a ajuda de variáveis auxiliares.

7 Filtragem

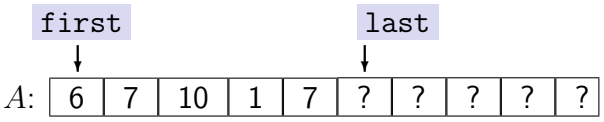
Descrição

Escreva uma função em C++ chamada `filter` que "filtra" os elementos no intervalo [`first`; `last`) definido sobre um vetor de inteiros por meio de ponteiros, retirando todos os **valores nulos e negativos** e preservando a ordem relativa dos elementos filtrados. A função deve retornar um ponteiro para a posição após o último elemento que permaneceu no vetor depois de realizada a operação de filtragem.

Considere o exemplo abaixo com apenas 10 elementos no intervalo:



depois de filtrado o intervalo fica com tamanho "lógico" $= 5$.



A função `filter` deve ter a seguinte assinatura:

```
const int *filter( const int * first, const int * last );
```

Note que em nenhum momento a memória original do vetor original é liberada ou realocada. A função apenas rearranja os valores dentro do vetor e retorna o endereço logo após o último valor válido dentro do vetor filtrado.

Conhecimentos necessários

Utilização de função, relação entre ponteiro e memória, ponteiros para memória constante, aritmética de ponteiros, laços, troca interna de valores em vetor.

Observação: Não é necessário criar um outro vetor para armazenar o resultado da filtragem. A filtragem **deve** ser feita *internamente*, ou seja, dentro do próprio vetor, com a ajuda de variáveis auxiliares.

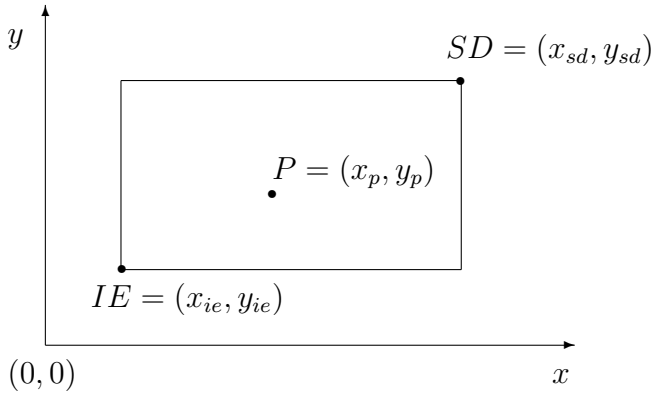
8 Ponto em Retângulo 1

Descrição

Considerando a estrutura abaixo

```
struct Ponto {
    int x; //!< coordenada X do ponto.
    int y; //!< coordenada Y do ponto.
    /// Construtor padrão com argumentos default.
    Ponto(int xi=0, int yi=0 ) : x{xi}, y{yi} // copia args p/ campos.
    {/* empty */}
};
```

para representar as coordenadas Cartesianas de um ponto no plano bidimensional (2D), implemente uma função em C++ que verifica se um ponto $P = (x_p, y_p)$, determinado por suas coordenadas Cartesianas, está localizado **dentro**, **na borda** ou **fora** de um retângulo definho por dois pontos: o canto inferior esquerdo $IE = (x_{ie}, y_{ie})$ e o canto superior direito $SD = (x_{sd}, y_{sd})$.



A função deve receber 3 pontos como referências constante representando, respectivamente, os dois pontos, IE e SD, que definem um retângulo, e o ponto a ser testado P. Assuma que os pontos IE e SD definem um retângulo válido, i.e $IE \neq SD$ (pelo menos uma coordenada de um ponto é diferente do outro).

A seguir, a função deve realizar testes e indicar se o ponto P está *dentro*, *na borda* ou *fora* do retângulo, retornando, respectivamente, os valores 0, 1 ou 2 para cada caso ora descrito.

Se desejar tornar seu código mais inteligível, você pode utilizar a enumeração abaixo

```
enum location_t: int { INSIDE=0, BORDER=1, OUTSIDE=2 };
```

A função pode ter a assinatura abaixo:

```
int pt_in_rect( const Ponto& R1, const Ponto& R2, const Ponto& P );
```

ou a assinatura seguinte em caso de uso da enumeração definida acima:

```
location_t pt_in_rect( const Ponto& R1, const Ponto& R2, const Ponto& P );
```

Conhecimentos necessários

Utilização de função, tipos heterogêneos (struct), passagem de parâmetro por referência constante, enumeração, Geometria 2D, condicionais, expressões lógicas.

Curiosidade: O algoritmo de ponto-em-retângulo é muito importante e deve ser bem eficiente, visto que ele pode ser utilizado, por exemplo, toda vez que você usa o *touch screen* do seu celular. O seu dedo é o *ponto* do problema e objeto que você seleciona, como por exemplo as teclas de um teclado virtual de entrada, é essencialmente modelado como um *retângulo*.

9 Ponto em Retângulo 2

Descrição

Implemente um programa em C++ que recebe da entrada padrão um número indeterminado de linhas, cada uma correspondendo a um caso de teste. Cada caso de teste contém informações correspondentes a um possível retângulo e um ponto, ambos definidos no plano Cartesiano 2D. Para cada caso de teste o programa deve (1) verificar se o retângulo é valido e, em caso verdadeiro (2) classificar o ponto em relação ao retângulo em uma das três possibilidades: *fora*, *na fronteira*, ou *dentro* do retângulo.

Um retângulo é considerado válido se e somente se pelo menos uma das quatro coordenadas dos vértices que o define for diferente, ou seja $R_1 \neq R_2$. Portanto, o programa deve aceitar os chamados retângulos "degenerados" que formam uma linha vertical ou horizontal, como por exemplo: $IE(2,5) \times SD(2,7)$ ou $IE(-53,-4) \times SD(-5,-4)$.

A classificação do posicionamento do ponto em relação ao retângulo **deve** ser feita através da invocação da função implementada na Questão 8. Lembre que para a função `pt_in_rect()` funcionar corretamente é necessário informar como argumentos as coordenadas do canto inferior esquerdo e superior direito. Portanto, seu programa deve analisar os vértices de entrada e fazer os ajustes necessários (por exemplo, criando novos pontos a partir das coordenadas originais) para satisfazer o pré-requisito da função de classificação.

Formato de Entrada/Saída

Cada linha lida da entrada padrão deve corresponder a um caso de teste, tendo o formato: $x_1\ y_1\ \ x_2\ y_2\ \ x_3\ y_3$, sendo $-1000 \leq x_i, y_i \leq 1000$. Os quatro primeiros valores representam as coordenadas de dois vértices quaisquer do retângulo: $R_1 = (x_1, y_1)$ e $R_2 = (x_2, y_2)$. Note que esses dois vértices podem representar qualquer um dos quatro possíveis cantos de um retângulo: inferior esquerdo, inferior direito, superior esquerdo, ou superior direito. Os últimos dois valores representam as coordenadas do ponto, $P = (x_3, y_3)$, a ser testado contra o retângulo definido na mesma linha.

Se o retângulo for inválido o programa deve enviar para a saída padrão a mensagem *"invalid"* (sem as aspas) e processar o próximo caso de teste (se ainda existir algum).

No caso de retângulo válido o seu programa deve enviar para a saída padrão uma das possíveis mensagens (sem aspas) abaixo, conforme o caso:

- *"inside"*, se o ponto P estiver totalmente dentro do retângulo;
- *"border"*, se o ponto P estiver sobre alguma das bordas do retângulo, e;
- *"outside"*, se o ponto P estiver localizado totalmente fora do retângulo

Exemplo de entrada 1:

```
-3 -1 3 1 0 0
2 2 9 7 4 2
7 9 2 2 2 7
4 5 4 5 -1 -2
-3 0 5 5 1 6
-3 0 5 5 6 4
1 2 -5 7 -1 6
3 7 -2 -2 2 2
```

Saída correspondente 1:

```
inside
border
border
invalid
outside
outside
inside
inside
```

Conhecimentos necessários

Utilização de função, reutilização de código, tipos heterogêneos (struct), passagem de parâmetro por referência constante, enumeração, Geometria 2D, condicionais compostos, expressões lógicas.

⌣ FIM ⌣