Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • DIM0120

◁ Trabalhando com Intervalos sobre Vetor ▷

20 de agosto de 2018

1 Introdução

Esta lista de exercício tem por objetivos oferecer uma prática para *(i)* demonstrar a relação próxima entre **ponteiros** e **vetores**; em particular, estamos interessados em utilizar ponteiros para passar valores por referência e para acessar elementos ao longo de um bloco contínuo de memória via **aritmética de ponteiros**; e *(ii)* introduzir o conceito de **intervalos** ou *ranges* de trabalho definidos sobre containers (no nosso caso, vetores); em particular, deseja-se introduzir o uso de intervalos do tipo [`first`; `last`) como mecanismo para definir dados sobre os quais uma determinada função deve atuar. Este mecanismo é usado extensivamente por bibliotecas profissionais, como as funções definidas no cabeçalho `<algorithms>` do STL.

O aprendizado deste conteúdo serve de preparação para introduzir o conceito de **iteradores**, que descreve tipos que podem ser usados para identificar e percorrer os elementos de um container de dados.

2 Exercícios

1. Desenvolva uma função `negate` que nega o sinal de todos os elementos no intervalo [`first`, `last`) definido sobre um vetor de inteiros. Assuma que o intervalo passado para a função é válido. O protótipo da função `negate` é:

```
void negate( int * first, int * last );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para negar.

Confira abaixo como seria diversas chamadas a `negate` no código cliente:

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
int main() {
    int Vet[] = {1, 2, -3, -4, 5, -6};

    // Nega todo o vetor.
    negate( std::begin(Vet),   std::end(Vet) );
    // Nega do 3ro até o último elemento do vetor.
    negate( std::begin(Vet)+2, std::end(Vet) );
    // Nega apenas o 4to elemento do vetor.
    negate( std::begin(Vet)+3, std::begin(Vet)+4 );
```

```
    return 0;
}
```

2. Desenvolva uma função `min` que retorna um ponteiro para a primeira ocorrência do **menor** elemento no intervalo [`first`, `last`) definido sobre um vetor de inteiros. Assuma que o intervalo passado para a função é válido. O protótipo da função `min` pode ser:

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

- `first`, `last` - ponteiros que definem o intervalo de elementos para examinar.

Confira abaixo como seria o uso de `min` no código cliente:

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
    int A[] = {1, 2, -3, -4, 5, -6};

    // Deveria imprimir -6.
    auto result = min( begin(A), end(A) );
    std::cout << *result << std::endl;

    // Deveria imprimir -4
    result = min( begin(A)+1, begin(A)+5 );
    std::cout <<  *result << std::endl;

    return 0;
}
```

3. Desenvolva uma função `reverse` que inverte a ordem dos elementos no intervalo [`first`, `last`) definido sobre um vetor de inteiros. Assuma que o intervalo passado para a função é válido. Procure utilizar a função `std::swap` para realizar as trocas. O protótipo da função `reverse` pode ser:

```
void reverse( int *first, int *last );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para inverter.

Confira abaixo um exemplo de como seria o uso da função `reverse` pelo código cliente.

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
    int A[] = { 1, 2, 3, 4, 5 };

    // aplicar reverse sobre todo o vetor.
    reverse( begin(A), end(A) );
```

```
    // O comando abaixo deveria imprimir A com o conteúdo 5, 4, 3, 2, 1
    for( auto i( begin(A) ) ; i != end(A) ; ++i )
        cout << *i << " ";
    cout << std::endl;

    // aplicar reverse sobre parte do vetor.
    reverse( begin(A)+1, begin(A)+4 );

    // O comando abaixo deveria imprimir A com o conteúdo 5, 2, 3, 4, 1
    for( auto i( begin(A) ) ; i != end(A) ; ++i )
        cout << *i << " ";
    cout << std::endl;

    return 0;
}
```

4. Desenvolva uma função `scalar_multiplication` que multiplica um inteiro passado por parâmetro por todos os elementos no intervalo [`first`, `last`) definido sobre um vetor de inteiros. Assuma que o intervalo passado para a função é válido. O protótipo da função `scalar_multiplication` pode ser:

```
void scalar_multiplication( int *first, int *last, int scalar );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para inverter.
- `scalar` - valor scalar (inteiro) a ser multiplicado pelos elementos do intervalo.

Confira abaixo um exemplo de como seria o uso e resultado da função `scalar_product`.

```
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
    int Vet[] = {1, 2, -3, -4, 5, -6};

    scalar_multiplication( begin(Vet),  end(Vet), 3 );
    // O vetor resultante seria:
    // { 3, 6, -9, -12, 15, -18 }.

    return 0;
}
```

5. Desenvolva uma função `dot_product` que calcula e retorna o **produto escalar** entre 2 vetores (entidade matemática, não arranjo). O produto escalar entre dois vetores $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ e $\mathbf{b} = [b_1, b_2, \ldots, b_n]$, é definido como:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

onde $\sum$ representa somatório e n é o comprimento (dimensão) do espaço vetorial. O protótipo da função `dot_product` pode ser:

```
int dot_product( const int *a_first, const int *a_last, const int *b_first );
```

- `a_first`, `a_last` - ponteiros que definem o intervalo correspondente ao vetor $\mathbf{a}$ da operação.
- `b_first` - ponteiros para o início do intervalo correspondente ao vetor $\mathbf{b}$ da operação.

Assuma que os intervalos fornecidos para a função são válidos. De acordo com a definição de produto escalar, os dois vetores **devem** ter o mesmo comprimento. Portanto, para acessar o segundo vetor basta saber onde ele inicia, `b_first`, que é possível deduzir onde ele termina com base no comprimento do vetor $\mathbf{a}$. O comprimento de $\mathbf{a}$ = `std::distance(a_first, a_last)` pode ser obtido por meio da função `std::distance`, que é uma função da biblioteca `<iterator>` do STL. Confira abaixo um exemplo de como seria o uso e resultado da função `dot_product`.

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
    // Desejamos multiplicar a=[1, 3, -5] · b=[4, -2, -1].
    int Vet[] = {1, 3, -5, 4, -2, -1 }; // Os dois vetores armazenados no mesmo arranjo.

    auto result = dot_product( begin(Vet),  begin(Vet)+3, // Vetor a
                               begin(Vet)+3 );             // Vetor b

    // O comando abaixo deveria imprimir 3, se correto.
    cout << ">>> O resultado é: " << result << std::endl;
    // [1,3,-5]·[4,-2,-1] = (1)(4)+(3)(-2)+(-5)(-1)
    //                    = 4 - 6 + 5
    //                    = 3.

    return 0;
}
```

6. Desenvolva uma função `compact` que "compacta" os elementos no intervalo [`first`, `last`) e retorna um ponteiro para o elemento **imediatamente posterior** ao último elemento compactado que permanecerá no intervalo. O processo de *compactação* consiste em mover para o início do intervalo todos os elementos positivos e não nulos, mantendo a ordem relativa entre os elementos originais. Desta forma, todos os elementos nulos ou negativos são eliminados no processo. Considere o exemplo abaixo com apenas 10 elementos, assumindo que `first = begin(A)` e `last=end(A)`:

A:	-2	-8	2	7	-3	10	1	0	-3	7
	0	1	2	3	4	5	6	7	8	9

depois de compactado a função deveria retornar o novo `last`, ou seja, um ponteiro para o endereço da posição 5 do vetor, portanto, após o último elemento compactado, que foi o 7.

A:	2	7	10	1	7	10	1	0	-3	7
	0	1	2	3	4	5	6	7	8	9

7. Desenvolva uma função `copy` que copia os elementos no intervalo [`first`, `last`) para outro intervalo iniciando em `d_first`. É precondição que o intervalo destino é capaz de receber a cópia dos elementos, i.e. existe a garantia de haver espaço suficiente no vetor destino. A função retorna um ponteiro para o elemento no intervalo destino logo após o último elemento copiado. O protótipo da função `copy` pode ser:

```
int * copy( const int *first, const int *last, int *d_first );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para copiar.
- `d_first` - ponteiro para o início do intervalo destino da cópia.

Confira abaixo um exemplo de como seria o uso da função `copy` pelo código cliente.

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
    int A[] = { 1, 2, 3, 4, 5 }; // Vetor "fonte".
    int B[] = { 0, 0, 0 }; // Vetor "destino".

    // Copiar elementos 2, 3, e 4 para o início de B.
    auto b_last = copy( begin(A)+1, begin(A)+4, begin(B) );

    // O comando abaixo deveria imprimir B com o conteúdo 2, 3, 4.
    for( auto i( begin(B) ) ; i != b_last ; ++i )
        cout << *i << " ";
    cout << std::endl;

    return 0;
}
```

8. Desenvolva uma função `unique` que elimina repetições de elementos no intervalo [`first`, `last`), preservando a ordem relativas dos elementos **únicos** que sobrarem, e retorna um ponteiro após-o-final do intervalo resultante. O protótipo da função `unique` pode ser:

```
int * unique( int *first, int *last );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para examinar.

Confira abaixo um exemplo de como seria o uso da função `unique` pelo código cliente.

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
    int A[] = { 1, 2, 1, 2, 3, 3, 1, 2, 4, 5, 3, 4, 5 };

    // aplicar unique sobre A
    auto last = unique( begin(A), end(A) );
```

```
    // O comando abaixo deveria imprimir A com o conteúdo 1, 2, 3, 4, 5.
    for( auto i( begin(A) ) ; i != last ; ++i )
        cout << *i << " ";
    cout << std::endl;

    // Mostra o novo tamanho de A, que seria 5.
    std::cout << ">>> O comprimento (lógico) de A após unique() é: "
              << std::distance( begin(A), last) << "\n";

    return 0;
}
```

9. Considere que os elementos no intervalo [`first`, `last`) podem ser classificados como de cor PRETA (valor 0) ou BRANCA (valor 1) e estão disposta em uma ordem qualquer. Desenvolva uma função `sort_marbles` que *rearranja* os elementos do intervalo usando **apenas** a operação de *troca* (`std::swap`) entre elementos, de maneira que todas as ocorrências PRETA apareçam antes de todas ocorrências BRANCA, e retorna um pointeiro (iterador) apontando para o limite das regiões, ou seja, apontando para o início da região dos elementos de cor BRANCA.

A Figura 1 ilustra este processo para um vetor com $N = 7$ elementos. Em particular, se não houver elementos de cor BRANCA no vetor, o algoritmo retorna `last`; similarmente, se não houver elementos de cor PRETA no vetor, o algoritmo retorna `first`.

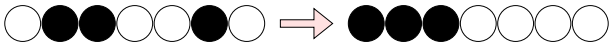

Figura 1: Rearranjando um vetor (à esquerda) de maneira a separar região BRANCA e PRETA (à direita). Neste exemplo o algoritmo deveria retornar um ponteiro para a posição 3, assumindo que o intervalo compreende todo o vetor (i.e. a partir de zero).

O protótipo da função `sort_marbles` pode ser:

```
int * sort_marbles( int *first, int *last );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para ordenar.
- retorna um ponteiro para o início da região das bolas BRANCAS.

Confira abaixo um exemplo de como seria o uso da função `sort_marbles` pelo código cliente.

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()

enum ball_t { B=0, W=1 }; // Black and White.
int * sort_marbles( int *first, int *last );
int main() {
    ball_t A[] = { W, B, B, W, W, B, W };  // input
    auto size_A = sizeof(A)/sizeof(ball_t);
    ball_t A_sorted[] = { B, B, B, W, W, W, W }; // expected output.
```

```
    auto result = sort_marbles( std::begin(A), std::end(A) );

    // White marbles shoud start at position 3 whithin the array.
    assert( std::distance(std::begin(A), result ) == 3 );
    // Validate answer
    for( auto i(0u) ; i < size_A ; ++i )
        assert( A[i] == A_sorted[i] );

    return 0;
}
```

10. Considere que [`first` , `last`) define um intervalo sobre um vetor de inteiros e que `pivot` é um ponteiro para um elemento dentro deste intervalo. Desenvolva uma função `partition` que *rearranja* (movimenta) os elementos do intervalo [`first` , `last`) com base em no valor apontado por `pivot` da seguinte forma: os elementos menores que o `pivot` aparecem no início do intervalo (em qualquer ordem), seguido dos elementos iguais ao valor do `pivot` , seguido dos elementos maiores que o `pivot` (em qualquer ordem). Por exemplo, se o vetor fornecido for $[\,-5,\ 7,\ 10,\ 7,\ 8,\ 9,\ 1,\ 7,\ -2,\ 3\,]$ com `pivot` apontando para a posição $= 3$ do vetor, uma possível saída seria $[\,-5,\ 3,\ -2,\ 1,\ 7,\ 7,\ 7,\ 9,\ 8,\ 10\,]$.

O protótipo da função `partition` pode ser:

```
void partition( int *first, int *last, int *pivot );
```

- `first` , `last` - ponteiros que definem o intervalo de elementos para examinar.
- `pivot` - aponta para o elemento-referência dentro do intervalo usado para rearranjar os elementos do vetor.

11. Desenvolva uma função `rotate` que realiza uma **rotação à esquerda** nos elementos do intervalo [`first` , `last`), preservando a ordem relativas dos elementos. O protótipo da função `rotate` pode ser:

```
void rotate( int *first, int *n_first, int *last );
```

- `first` - ponteiro para o início do intervalo original.
- `n_first` - ponteiro para o elemento que **deve** aparecer no início do intervalo rotacionado.
- `last` - ponteiros para o final do intervalo original.

O processo de rotação consiste em trocar os elementos do intervalo [`first` ; `last`) de tal maneira que o elemento `n_first` se torna o primeiro elemento do intervalo rotacionado e o elemento `n_first-1` se torna o último elemento do intervalo rotacionado. Confira abaixo um exemplo de como seria o uso da função `rotate` pelo código cliente.

```
#include <iostream>  // std::cout, std::endl
#include <iterator>  // std::begin(), std::end()
using namespace std;
int main() {
```

```
    int A[] = { 1, 2, 3, 4, 5, 6, 7 };

    // aplicar rotate, de maneira que 3 passe a ser o novo "primeiro" elemento em A.
    rotate( begin(A), begin(A)+2, end(A) );

    // O comando abaixo deveria imprimir A com o conteúdo 3, 4, 5, 6, 7, 1, 2
    for( auto i( begin(A) ) ; i != end(A) ; ++i )
        cout << *i << " ";
    cout << std::endl;

    return 0;
}
```

∽ FIM ∽