Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030
– 1ª Avaliação, 8 de setembro de 2015 –

A PROVA TEM A DURAÇÃO DE **100 MINUTOS**.
LEIA ATENTAMENTE AS INSTRUÇÕES ABAIXO ANTES DE INICIAR A PROVA

- ⋆ *Esta é uma prova individual e com consulta restrita.*
- ⋆ *Você terá 100 minutos para a realização da prova. As respostas devem ser fornecidas na forma de um arquivo digital compactado contendo seus programas.*

1 Apresentação

Nesta avaliação você deve implementar duas classes em C++ e funções template. Todas as definições das classes e protótipos de funções devem ser escritas em um arquivo cabeçalho denominado `geometry.h`. Já a implementação dos métodos das classes e funções devem ser escritas em um arquivo de implementação denominado `geometry.cpp`. Se alguns métodos for de apenas uma linha (*in-liners*) você tem a opção de deixá-los no arquivo cabeçalho ou no arquivo de implementação.

Já o `main()` para testar as classes e o funcionamento das funções template devem estar em um arquivo `search_poly.cpp`.

Ao finalizar a avaliação você deve submeter uma pasta compactada contento, pelo menos, estes três arquivos: `geometry.h`, `geometry.cpp` e `search_poly.cpp`. Você pode organizar seus arquivos em pastas separadas, se assim desejar.

2 Parte I: Implementar classe `Point`

Implemente uma classe representando um ponto bidimensional (2D) em um sistema de coordenadas Cartesianas.

2.1 Fundação

Criar a classe com dois inteiros privados, denominados de `x` e `y`.

2.2 Construtor

Implemente um único construtor que, se invocado sem argumentos, inicializa o ponto na origem do sistema de coordenadas, ou seja $(0, 0)$. Porém, se invocado com dois argumentos `x` e `y`, cria um ponto localizado na coordenada (x, y).
(*Dica:* Lembre-se dos argumentos default.)

2.3 Funções Membro

Sua classe deve suportar as seguintes operações usando as assinaturas abaixo:

- ⋆ Recupera a coordenada x do ponto
  `int Point::getX() const`
- ⋆ Recupera a coordenada y do ponto
  `int Point::getY() const`
- ⋆ Define a coordenada x do ponto
  `void Point::setX( const int newX )`
- ⋆ Define a coordenada y do ponto
  `void Point::setY( const int newY )`

3 PointArray

Nesta questão você deve implementar uma classe para representar um arranjo de `Point`. A classe deve suportar crescimento dinâmico do arranjo, e deve guardar seu comprimento atual.

3.1 Fundação

Você deve criar a classe com dois membros privados, um ponteiro para o início do arranjo de `Point` e um inteiro para armazenar o comprimento (ou tamanho) atual do arranjo.

3.2 Construtor

O construtor padrão (sem argumentos) deve criar um arranjo de comprimento zero.

Implemente um segundo construtor que recebe como seus argumentos um vetor simples de `Point` chamado de `points` e um inteiro denominado de `size` correspondente a quantidade elementos no arranjo. Este construtor deve inicializar o objeto `PointArray` com o comprimento informado, copiando seus valores de `points`. Internamente você deve alocar a memória de `PointArray` para acomodar os valores passados.

`PointArray::PointArray( const Point points[], const int size )`

Finalmente, implemente um construtor cópia para um dado `PointArray`.

`PointArray::PointArray( const PointArray& pv )`

(*Dica*: cuidado com o problema de cópia rasa $\times$ cópia profunda!)

3.3 Destruidor

Implemente o método destruidor que libera a memória interna de `PointArray`.

`PointArray::~PointArray( )`

3.4 Métodos

Por se tratar de um arranjo dinâmico de `Point`, você deve implementar um método interno (privado) `PointArray::resize(int n)` que modifica o tamanho do arranjo para `n`, copia os primeiros `min`(tamanho anterior, `n`) elementos existentes para o novo arranjo e libera a memória do arranjo antigo.

As demais funções públicas são:

1. Adicionar um `Point` no final do arranjo.
   `void PointArray::push_back( const Point &p )`
2. Inserir um `Point` em uma posição arbitrária do arranjo, `position`, deslocando para direita os elementos após `position` para criar espaço para o novo elemento.
   `void PointArray::insert( const int position, const Point &p )`
3. Remover um `Point` em uma posição arbitrária do arranjo, `position`, deslocando para esquerda os elementos restantes do arranjo.
   `void PointArray::remove( const int position )`
4. Recuperar o comprimento do arranjo.
   `const int getSize() const`
5. Remover todos os elementos do arranjo e definir seu tamanho para zero.
   `void PointArray::clear()`
6. Retornar ou atribuir um `Point` em uma posição arbitrária do arranjo, assumindo que a posição inicial é zero, como em vetores. Neste caso você deve sobrescrever o operador de acesso via colchetes.
   `const Point PointArray::operator[]( const int position ) const`
   `Point& PointArray::operator[]( const int position )`

Para os métodos de remoção e inserção de `Points` você deve ajustar a memória de maneira que ela sempre corresponda ao número exato de elementos armazenados.

Para todos os métodos de `PointArray` que trabalham com uma posição arbitrária dentro do arranjo, você deve primeiramente verificar se a posição passada como argumento é válida. Se não for válida, você deve lançar uma exceção da seguinte forma:

```
throw std::out_of_range("Indice fornecido fora da faixa valida!");
```

Para tanto você deve incluir o cabeçalho `<stdexcept>` em `geometry.h`.

4 Busca

Considere que um `PointArray` representa um *polígono* fechado, ou seja, Cada dois pares consecutivos de pontos representam uma aresta de um polígono e o último ponto se liga ao primeiro. Desta forma, dado um polígono $P = \{\ p_1 : (x_1, y_1)\ ;\ p_2 : (x_2, y_2)\ ;\ p_3 : (x_3, y_3)\ \}$ com 3 pontos podemos representar um triângulo cujos lados são $\overline{p_1\, p_2}$, $\overline{p_2\, p_3}$ e $\overline{p_3\, p_1}$.

No programa `search_poly.cpp` você deve criar um arranjo `P` com, pelo menos, 5 polígonos. Você deve escrever uma função para fazer uma busca linear em `MyPolys` por um polígono-alvo `PolyTarget`. Sua função deve dar suporte a pelo menos dois métodos diferentes de identificação de igualdade entre polígonos, da seguinte forma:

- ⋆ **Método #1**: Um polígono a é igual a um polígono b se a e b têm a mesma quantidade de pontos, tais pontos possuem a mesma coordenada Cartesiana e estão dispostos na mesma ordem interna dentro do arranjo.
- ⋆ **Método #2**: Um polígono a é igual a um polígono b se a e b têm a mesma quantidade de pontos, tais pontos possuem a mesma coordenada Cartesiana e os pontos estão dispostos em uma *ordem qualquer* dentro do arranjo.

Por exemplo, considerando

$$a = \{(1,0);(2,2);(3,-4)\},$$
$$b = \{(1,0);(3,-4);(2,2)\},$$
$$c = \{(1,0);(2,2);(3,-4);(8,8)\} \text{ e}$$
$$d = \{(1,0);(2,2);(3,-4)\}$$

sob o critério de igualdade do Método $\#1$, apenas a e d são iguais. Por outro lado, sob o critério de igualdade do Método $\#2$, são considerados iguais os políginos a, b e d.

$\sim$ FIM $\sim$