2022_1 - PROGRAMAÇÃO E DESENVOLVIMENTO DE SOFTWARE II - TA_TN - METATURMA

L01E04 - Drones (3,0 pts)

Data de entrega: terça, 17 Mai 2022, 23:59
Arquivos requeridos: Ponto2D.hpp, Ponto2D.cpp, Drone.hpp, Drone.cpp (Baixar)
Tipo de trabalho: Trabalho individual

Neste exercício você deverá implementar dois **TADs** para uma equipe de robótica. Utilizando **Structs**, você deve criar os seguintes TADs: **Ponto2D** e **Drone**. Cada TAD deve seguir as especificações abaixo:

- **Ponto2D**:

1. `Ponto2D(double, double)`: método construtor que recebe dois parâmetros do tipo double que representam as coordenadas (x,y) e devem ser armazenados internamente. Atenção: veja a Dica 1.
2. `double calcular_distancia(Ponto2D* ponto)`: método que calcula a distância euclidiana para outro ponto no plano.
3. `std::string get_dados()`: método que retorna uma string com os dados no seguinte formato: "**[x, y]**". Os valores de x e y devem possuir no máximo **2 casas decimais de precisão** (veja o primeiro link nas referências abaixo e o arquivo main.cpp).

- **Drone:**

1. Atributos: Além de outros atributos que você julgar necessário, o TAD também deve ter um atributo '**_energia**', do tipo double, que é sempre inicializado com valor 100.
2. `Drone(int, Ponto2D , double)`: método construtor que recebe três parâmetros: (i) um inteiro é o id único do drone, (ii) uma variável do tipo Ponto2D (atenção, não deve ser usado ponteiro!), e (iii) um double, que representa o raio de comunicação do drone.
3. `void mover(double v, double th, double t)`: método que atualiza a posição do drone nos eixos x e y de acordo com os parâmetros passados: v (magnitude do vetor velocidade), th (orientação, em radianos, do vetor velocidade), t (o tempo que a velocidade foi aplicada). Após o deslocamento, a distância percorrida deve ser deduzida do campo '_energia'. Caso o valor fique menor ou igual a 50 deve-se imprimir a seguinte mensagem: "Alerta, energia baixa!". Dica: faça uma decomposição vetorial simples e use a equação de cinemática.
4. `double calcular_distancia(Drone* drone)`: método que calcula e retorna a distância euclidiana para outro drone passado como parâmetro.
5. `void broadcast_mensagem(Drone** drones, int n)`: método que recebe um array de ponteiros para drones e um parâmetro que informa a quantidade de elementos nesse array. Envia uma mensagem apenas para os drones que estão à uma distância menor ou igual o raio de comunicação. A mensagem é uma string no seguinte formato: "**Drone: id, Posição: [x, y]**", onde id é o id do drone enviando a mensagem e x, y a sua posição atual.
6. `void salvar_mensagem(string mensagem)`: método que salva internamente a mensagem recebida por um drone. A mensagem deve ser salva em um 'buffer' interno que possui apenas 5 posições. Além disso, a inserção deve ser feita de maneira circular, ou seja, ao chegar na última posição a próxima mensagem deve ser salva na primeira posição novamente e assim sucessivamente (você deve pensar a forma de fazer isso). Dica: esse método deve ser chamado dentro de 'broadcast_mensagem'.
7. `imprimir_mensagens_recebidas()`: método que imprime todas as mensagens recebidas por um drone, ou seja, que estão no 'buffer'. A impressão segue o seguinte formato:
```
Mensagens de {id_proprio}:
Drone: {id_mensagem}, Posição: [x, y]
Drone: {id_mensagem}, Posição: [x, y]
```
8. `void imprimir_status()`: imprime a situação atual do drone no seguinte formato: "**id x y energia**". Atenção, nesse caso utilize **tab** (\t) para separar os elementos.

Como entrada serão fornecidos a posição (x, y) e o raio de comunicação de cada drone. Em seguida, o sistema é executado considerando os comandos informados e algumas informações devem ser impressas na tela.

Nesse exercício **você não precisa implementar a função main**! O objetivo é simular o caso em que você está implementando um TAD que será

utilizado por outra pessoa (que conhece apenas o contrato). Dessa forma, a leitura dos dados de entrada e jogadas já estão implementadas, e **você deve apenas implementar os TADs** (Ponto2D e Drone). Você é livre para implementar nos TADS quaisquer outros métodos não mencionados que julgar necessário.

Para ilustrar, abaixo são apresentados exemplos de entrada/saída.

Nesse primeiro exemplo é informado apenas um drone. O comando 's' é usado para imprimir o status do drone e o comando 'm' move o drone.

```
input=
1
0.0 0.0 1.0
s
m 0 1.0 0.0 2.0
s
output=
0 0.00 0.00 100.00
0 2.00 0.00 98.00
```

No segundo exemplo são informados dois drones. O comando 'b' é usado para informar que o drone 0 (primeiro informado) vai fazer um broadcast e o comando 'p' informa que deve-se imprimir as mensagens recebidas pelo drone 1 (segundo informado). Em seguida, o drone 0 se movimenta e o drone 1 fica fora de alcance, por isso ao se fazer novo broadcast o buffer não deveria ser modificado.

```
input=
2
0.0 0.0 3.0
2.0 0.0 3.0
s
b 0
p 1
m 0 1.0 3.1415 2.0
s
b 0
p 1
output=
0 0.00 0.00 100.00
1 2.00 0.00 100.00
Mensagens de 1:
Drone: 0, Posição: [0.00, 0.00]
0 -2.00 0.00 98.00
1 2.00 0.00 100.00
Mensagens de 1:
Drone: 0, Posição: [0.00, 0.00]
```

Atenção: Lembre-se de fazer a correta modularização utilizando os arquivos **.hpp** e **.cpp**.

Dica1:
Deve-se conseguir criar variáveis do tipo Ponto2D sem passar nenhum valor para o construtor. Para isso, você pode criar um construtor vazio (um TAD pode ter vários construtores) ou utilizar valores padrão no método construtor com parâmetros (veja o último link nas referências abaixo).

Dica2:
Você pode copiar o arquivo main.cpp aqui para testar localmente e ver como todo o setup inicial é feito (entrada/saída, comandos, etc).

Referências:
https://www.cplusplus.com/reference/iomanip/setprecision/
https://www.cplusplus.com/reference/sstream/stringstream/
https://www.cplusplus.com/doc/tutorial/dynamic/
https://www.cplusplus.com/doc/tutorial/structures/
https://en.cppreference.com/w/cpp/language/default_arguments

VPL

◄ L01E03 - Ponteiros (2,0 pts)

Seguir para...

L01E05 - Fila atendimento (3,0 pts) ►