Trabalho Prático 1 - Tipos Abstratos de Dados

Luiz Felipe Matos Pedone[1]

[1] Universidade Federal de Minas Gerais

luiz.pedone@dcc.ufmg.br

1. Introdução

Este trabalho apresenta os métodos e discute os resultados da implementação do Tipo Abstrato de Dados (TAD) Ponto 3D e a sua aplicação em um caso específico. Neste TAD, é definido um ponto em três dimensões e operações elementares que podem ser executadas com pontos nesta dimensão.

Para a implementação deste TAD, foi utilizado o conceito de classes da linguagem de programação C++. Segundo a documentação referência desta linguagem, "classes são uma expansão do conceito de estrutura de dados, não contendo só dados, mas também funções para manipulação destes" (Classes, C++ Documentation).

Como método de avaliar os resultados da classe criada, foi implementada uma aplicação que recebe de um arquivo de texto alguns pontos tridimensionais e as operações que devem ser executadas com estes pontos e grava os resultados destas operações em um novo arquivo.

2. Metodologia

2.1. Classe Ponto3D: Características Gerais

Para a implementação do TAD Ponto 3D, foi utilizada a interface pública previamente disponibilizada da classe com as seguintes funções.

```
public:
    // Cria um ponto com coordenadas (0.0, 0.0, 0.0).
    Ponto3D();

    // Cria um ponto com coordenadas (x, y, z).
    Ponto3D(double x, double y, double z);

    // Cria um ponto com as mesmas coordenadas de p.
    Ponto3D(Ponto3D& p);

    // Retorna a coordenada x do ponto.
    double& x();

    // Retorna a coordenada y do ponto.
    double& y();

    // Retorna a coordenada z do ponto.
    double& z();

    // Faz com que as coordenadas do ponto corrente sejam igual as de p.
    void operator=(Ponto3D& p);

    // Translada o ponto em qualquer dos eixos.
    void Transladar(double dx, double dy, double dz);

    // Muda a escala do ponto em qualquer dos eixos.
    void MudarEscala(double fx, double fy, double fz);

    // Rotaciona o ponto 'ang' radianos em torno do eixo 'e'.
    // Os valores válidos para 'e' são 'X', 'Y', ou 'Z'.
    void Rotacionar(char e, double ang);
```

Tabela 1: Interface pública disponibilizada para a implementação do TAD.

Como interface privada da classe, foram definidos:

```
private:
    // Coordenada X do Ponto3D.
    double x_;
    // Coordenada Y do Ponto3D.
    double y_;
    // Coordenada Z do Ponto3D.
    double z_;
```

Tabela 2: Membros privados da classe Ponto3D.

O Ponto3D foi definido como três variáveis do tipo `double`, nomeadas como x_, y_ e z_. Foi testada a implementação com um vetor de três dimensões, porém este modo de implementação não demonstrou nenhuma melhoria na implementação das demais funções.

2.2. Funções

A classe Ponto3D possui dez funções, que vão desde a construção do ponto tridimensional até operações como translação, rotação e mudança de escala. Neste tópico, os métodos contidos na classe serão descritos e, quando necessário, comentados.

2.2.1. Construtores

Para a criação de um Ponto 3D, o usuário da classe deve utilizar um dos três construtores da classe: o construtor de um Ponto3D com coordenadas iguais a zero (0.0, 0.0, 0.0), o com as coordenadas X, Y e Z definidas ou um que copia um ponto já existe para um novo.

```
Ponto3D::Ponto3D(){
   this -> x_ = 0.0;
   this -> y_ = 0.0;
   this -> z_ = 0.0;
}
Ponto3D::Ponto3D(double x, double y, double z){
   this -> x_ = x;
   this -> y_ = y;
   this -> z_ = z;
}
Ponto3D::Ponto3D(Ponto3D &p){
       this -> x_ = p.x_;
       this -> y_ = p.y_;
       this -> z_ = p.z_;
}
```

Tabela 3: Construtores do Ponto3D.

2.2.1. Funções

Para executar operações com o tipo de dado Ponto3D, é necessário executar funções que saibam lidar com a estrutura definida na classe. Para isto, foram definidas algumas funções que executam operações fundamentais para a manipulação dos pontos. A lista das funções está na Figura 1. Nas próximas linhas, serão demonstrados todos as funções e, caso necessário, feitos alguns comentários sobre a implementação escolhida.

2.2.1. Funções de acesso às variáveis

Utilizando o conceito de encapsulamento, as propriedades do Ponto 3D foram definidas como privadas. Isto significa que o usuário que utiliza a classe não tem acesso a estas propriedades de forma direta. Este método de estruturação tem várias vantagens, como, por exemplo, a proteção da integridade das propriedades da classe.

No caso da classe Ponto 3D, para o usuário receber os pontos ele deve chamar as funções `double& x()`, `double& y()` e `double& z()`. Abaixo segue a implementação simples destas funções.

```
double& Ponto3D::x(){
   return x_;
}
double& Ponto3D::y(){
   return y_;
}
double& Ponto3D::z(){
 return z_;
}
```

Tabela 4: Funções de acesso às variáveis privadas.

2.2.2. Funções de manipulação dos dados

Para manipular os Pontos 3D gerados pela classe, foram criadas funções que realizam operações elementares com estes pontos. As funções da classe são: igualar o ponto vigente com um outro ponto, transladar o ponto, mudar a escala deste ponto e rotacionar o ponto.

2.2.2.1. Igualar o ponto vigente com um ponto já existente

Nesta função, o usuário pode definir um Ponto3D como a cópia de um Ponto3D já existente. Para facilitar o uso desta função, o operador igual foi sobrecarregado, tornando a operação similar as existentes nos tipos nativos da linguagem C++. Caso o usuário queira igualar apenas uma ou duas coordenadas, as funções `double& x()`, `double& y()` e `double& z()` podem ser utilizadas.

```
void Ponto3D::operator=(Ponto3D& p){
       this -> x_ = p.x_;
       this -> y_ = p.y_;
       this -> z_ = p.z_;
}
```

```
int main(){
//Ponto a definido como (3,2,1).
Ponto3D a (3,2,1);
//Ponto b é igual ao a. Logo, b=(3,2,1).
Ponto3D b = a;
}
```

Tabela 5: Implementação da função igual (esquerda) e demonstração da utilização da função (direita).

2.2.2.2. Transladar

A função transladar move o Ponto3D no espaço. Para utilizar esta função, o usuário deve passar os valores X, Y, Z que serão somados ou subtraídos das respectivas coordenadas do Ponto.

```
void Ponto3D::Transladar(double dx,
double dy, double dz){
       this -> x_ = x_ + dx;
       this -> y_ = y_ + dy;
       this -> z_ = z_ + dz;
}
```

```
int main(){
Ponto3D a (1,2,3);
a.Transladar(1.5, 2.4, 2.2);
// a=(2.5, 4.4, 5.2)
}
```

Tabela 6: Implementação da função transladar (esquerda) e demonstração da utilização da função (direita).

2.2.2.3. Mudar escala

Para a mudança de escala, multiplica-se as coordenadas do Ponto por um escalar definido para cada coordenada X, Y e Z.

```
void Ponto3D::MudarEscala(double fx,
double fy, double fz){
       this -> x_ = x_ * fx;
       this -> y_ = y_ * fy;
       this -> z_ = z_ * fz;
}
```

```
int main(){
Ponto3D a (1,2,3);
a.MudarEscala(1.5, 2.4, 2.2);
// a=(1.5, 4.8, 6.6)
}
```

Tabela 7: Implementação da função de mudança de escala (esquerda) e demonstração da utilização da função (direita).

2.2.2.3. Rotacionar o Ponto

Um ponto tridimensional pode ser rotacionado a partir de um dos eixos. Para executar esta operação, basta multiplicar o ponto por uma matriz de rotação. Para cada eixo, a matriz de rotação é:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Figura 1: Matrizes de rotação para cada um dos eixos.

Em cada matriz acima, é o ângulo em radianos em que o ponto será rotacionado em sentido anti-horário (Weisstein).

Para a utilização desta função, o usuário deve informar qual eixo será utilizado como referência e qual é o grau de rotação. A implementação desta função ficou da seguinte forma:

```
void Ponto3D::Rotacionar(char e, double arg){
    double aux[3];
    if(e == 'X'){
        aux[1] = y_*cos(arg)+z_*(sin(arg));
        aux[2] = y_*(-sin(arg))+z_*cos(arg);
        this -> y_ = aux[1];
        this -> z_ = aux[2];
    }
    if(e == 'Y'){
        aux[0] = x_*cos(arg)+z_*(-sin(arg));
        aux[2] = x_*sin(arg)+z_*(cos(arg));
        this -> x_ = aux[0];
        this -> z_ = aux[1];
    }
    if(e == 'Z'){
        aux[0] = x_*cos(arg)+y_*(-sin(arg));
        aux[1] = x_*(sin(arg))+y_*cos(arg);

        this->x_ = aux[0];
        this->y_ = aux[1];
    }
}
```

Tabela 8: Implementação da função de Rotacionar Ponto.

Inicialmente, uma implementação desta função com um vetor (Ponto3D) e três matrizes de Rotação (para os três eixos) foi feita. Porém, esta implementação não demonstrou nenhuma vantagem para este caso específico e foi descartada.

2.2. Aplicação com a classe

Como meio de testar a classe Ponto3D foi elaborada uma pequena aplicação. Esta aplicação lê um arquivo de texto com um número qualquer de pontos e executa operações também definidas no arquivo sobre estes pontos. Para a saída, um novo arquivo de texto é gerado com os resultados acumulados das operações.

2.2.1. Implementação

A implementação desta aplicação utiliza as seguintes bibliotecas além das já definidas pela classe Ponto3D:

- `<fstream>`: para a manipulação dos arquivos de entrada e saída;
- `<iostream>`: para a manipulação de entradas e saídas na tela;
- `<math.h>`: biblioteca com diversas definições e funções matemáticas.
- `<iomanip>`: para a manipulação da precisão da entrada e saída nos arquivos.

A aplicação lê um arquivo chamado "entrada.txt" com as seguintes características:

1. O número de pontos do arquivo;
2. As coordenadas dos pontos;
3. As operações a serem realizadas nestes pontos.

O arquivo de entrada deve ter o seguinte formato:

Descrição	**Arquivo: "entrada.txt"**
Número de Pontos	8
Coordenadas dos N pontos informados previamente.	0 0 0 15 0 0 15 15 0 0 15 0 0 0 15 15 0 15 15 15 15 0 15 15
Operações a serem realizadas sobre os pontos.	T -7.5 -7.5 0 R Z 30 E 3.0 3.0 3.0

Tabela 9: exemplo de arquivo de entrada.

A partir destes dados, a aplicação deve executar as operações sobre os pontos e armazenar os resultados acumulativos no arquivo "saida.txt".

Para isto, o programa foi implementado em quatro partes: (a) identificação do número de pontos a serem lidos; (b) leitura dos pontos; (c) execução das operações desejadas sobre os pontos; (d) inserção dos resultados no arquivo de saída. Para a parte (a), foi criada uma variável auxiliar para armazenar o número de pontos a serem lidos do arquivo. Com o número de pontos a serem lidos, é alocado um vetor Ponto3D dinamicamente. Em seguida, é executada a leitura dos pontos (b): um laço lê todos os pontos definidos pelo tamanho de (a). Dentro deste laço, é inserido os valores de cada ponto do arquivo em uma posição do vetor. A implementação foi feita da seguinte forma:

```
//Vetor do tipo Ponto3D onde todos os pontos do arquivo serão armazenados.
Ponto3D* vetor = new Ponto3D [size];
for(int i=0; i<size; i++){
        for(int j=0; j<3; j++){
            // Armazena coordenadas na variável temporária
            leArquivo >> tmp[j];
        }
        // Cria um Ponto3D auxiliar
        Ponto3D aux (tmp[0],tmp[1],tmp[2]);
        // Insere o Ponto3D na posição i do vetor
        vetor[i] = aux;
    }
```

Tabela 10: leitura dos pontos do arquivo e armazenamento no vetor.

Após a leitura de todos os pontos, o programa inicia a execução das operações. Diferentemente do caso da leitura de pontos, neste não há como definir o intervalo no qual o laço deve ser executado. Assim, enquanto houver valores a serem lidos no arquivo este laço será executado.

Como o primeiro valor de cada linha é o tipo de operação, três condições foram criadas para direcionar para cada operação. Abaixo segue a implementação da execução das operações.

```
while (leArquivo.good()){
        leArquivo >> value;
        // Operacao Transladar
        if(value == 'T'){
            leArquivo >> tmp[0];
            leArquivo >> tmp[1];
            leArquivo >> tmp[2];
            for(int j=0; j<size; j++){
                Ponto3D translada = vetor[j];
                translada.Transladar(tmp[0], tmp[1], tmp[2]);
                vetor[j] = translada;
            }
        }
        // Operacao Mudanca de Escala
        if(value == 'E'){
            leArquivo >> tmp[0];
            leArquivo >> tmp[1];
            leArquivo >> tmp[2];
            for(int j=0; j<size; j++){
                Ponto3D escala = vetor[j];
                escala.MudarEscala(tmp[0], tmp[1], tmp[2]);
                vetor[j] = escala;
            }
        }
        //Operacao Rotacionar Ponto
        if(value == 'R'){
            leArquivo >> eixo;
            leArquivo >> graus;
            for(int j=0; j<size; j++){
                Ponto3D rot = vetor[j];
                rot.Rotacionar(eixo, grauspararad(graus));
                vetor[j] = rot;
            }
        }
}
```

Tabela 11: execução das operações sobre os pontos.

A operação de rotacionar o ponto, a entrada recebida do arquivo é em graus. Porém, na interface pública da classe, a função Rotacionar recebe o ângulo de rotação em radianos. Para isto, foi necessário converter o ângulo de graus para radianos antes de executar a função.

Por fim, os resultados foram inseridos no arquivo “saída.txt. Para a inserção com a precisão de três casas decimais, foi utilizada a função setprecision, disponível na biblioteca <iomanip>. O arquivo de saída deve ter a seguinte formatação:

Descrição	Arquivo: “saida.txt”
Número de Pontos contidos no arquivo	3
Coordenadas dos pontos após as operações	-8.236 -30.736 0.000 30.736 -8.236 0.000 8.236 30.736 0.000

Tabela 12: exemplo de arquivo de saída da aplicação.

2.3. Compilador utilizado

Para compilar o programa desenvolvido, foi utilizado o compilador padrão GCC utilizando a biblioteca libstdc++. Esta biblioteca permite a utilização de classes exclusivas do C++, como a <iostream>.

3. Conclusões

Neste trabalho foram desenvolvidas habilidades de implementação de um TAD e utilização deste em uma aplicação prática. Desta forma, o trabalho além de permitir um entendimento de como estruturar os dados dentro da classe, permitiu também testar e avaliar as vantagens e desvantagens da implementação do Ponto3D.

Pelo fato de a interface pública da função não poder ser alterada, a função Rotacionar, por exemplo, necessitou de uma função auxiliar desenvolvida pelo desenvolvedor da aplicação para ser utilizada. Caso a interface pública pudesse ser editada, talvez fosse interessante adicionar o parâmetro unidade de medida, permitindo o usuário definir qual tipo de entrada: graus ou radianos.

Além dos pontos acima citados, o trabalho também permitiu a prática na implementação de aplicação manipulando arquivos e aprimoramento da lógica de programação.

Referências

Classes. Documentação da Linguagem de Programação C++. Em: <http://www.cplusplus.com/doc/tutorial/classes/>. Acessado em 24/10/2011.

Weisstein, Eric W. "Rotation Matrix." From MathWorld--A Wolfram Web Resource. Em: <http://mathworld.wolfram.com/RotationMatrix.html>. Acessado em 26/10/2011.