



Residência
em Software

Métodos e Sobrecarga em C++

Professores:
Álvaro Coelho, Edgar Alexander, Esbel
Valero e Hélder Almeida

INSTITUIÇÃO EXECUTORA



COORDENADORA

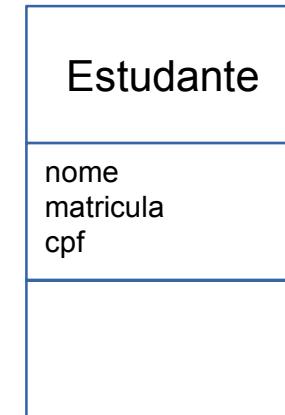


APOIO



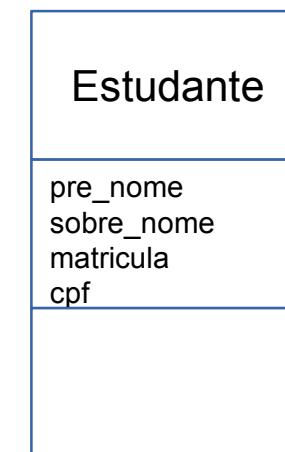
Métodos

- Um cenário
- Suponha a classe estudante (ao lado)
 - Observe o atributo Nome
- É o nome do estudante
 - Diversos artefatos (outros programas, funções, sistemas, etc.) utilizam este atributo
 - ex. Estudante e; cin >> e.nome; e.nome = z.nome; ...
 - Qual o problema?



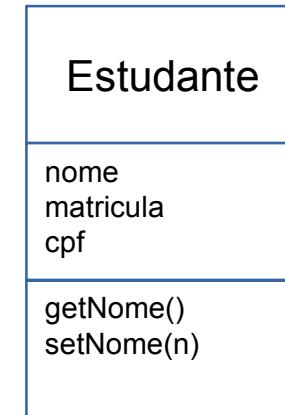
Métodos

- Qual o problema?
- Suponha que a classe mude
 - Prenome e Sobrenome
- O que acontece com todos os artefatos?



Alternativa

- É proibido acessar o atributo nome!
 - Use `getNome()` e `setNome(n)`
- Ex.
 - Estudante e;
 - ...
 - `cin >> x; e.setNome(x);`
 - `e.setnome(z.getnome());`
- Qual a vantagem?



Gerenciando mudança

- A mudança na classe Estudante não impacta as demais classes
 - getnome() agora retorna o nome (concatenado)
 - setnome(s) agora insere o pre e o sobrenome
 - Surgem métodos para ler e setar prenomes e sobrenomes
- Os demais artefatos não sofrem com a alteração da classe Estudante
- Diminuição do Acoplamento!

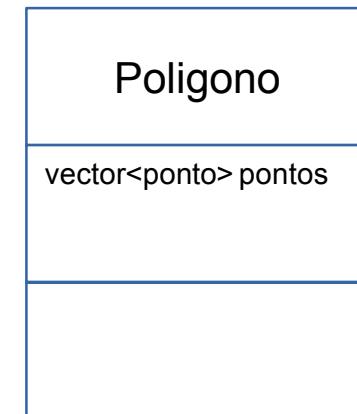
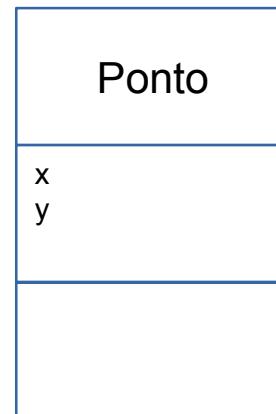
Estudante

pre_nome
sobre_nome
matricula
cpf

getnome()
setnome(s)
getpre_nome()
getsobre_Nome()
setpre_nome(s)
setsobre_Nome(s)

Exercício

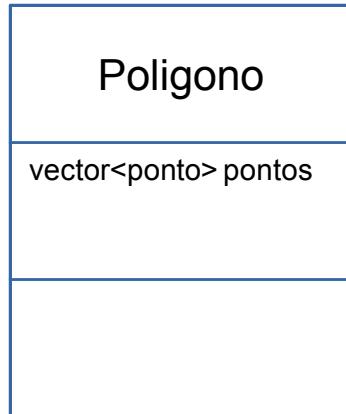
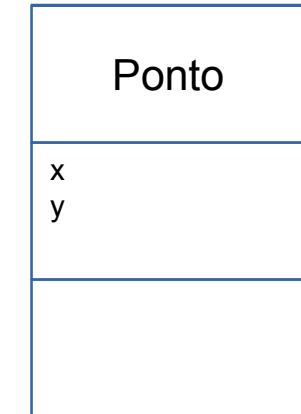
- Crie as classes abaixo
- Crie um programa que permita criação de um polígono* com seus vários pontos
- E depois liste todos os pontos do polígono



*Considere que os pontos serão inseridos de forma que SEMPRE formarão um polígono



Uma solução



```
#include <iostream>
#include <vector>

using namespace std;

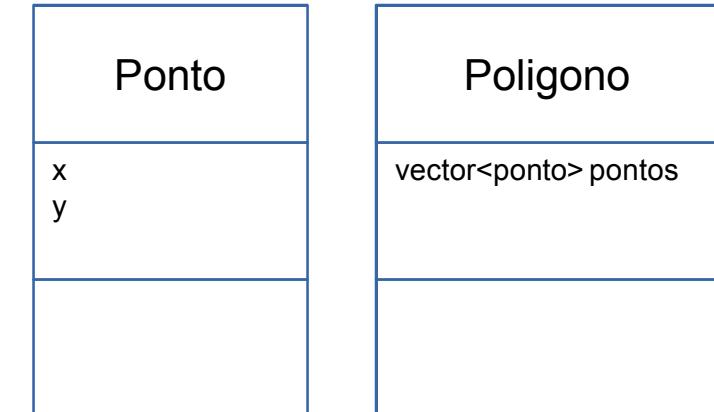
class Ponto {
public:
    float x, y;
};

class Poligono {
public:
    vector<Ponto> pontos;
};

int main () {
    Poligono poli;
    cout << "Criando um polígono!" << endl;
    char sn;
    do {
        cout << "Digite as coordenadas do ponto: ";
        Ponto p;
        cin >> p.x >> p.y;
        poli.pontos.push_back(p);
        cout << "Deseja inserir mais pontos (s/n)?";
        cin >> sn;
    } while (sn!='n');
    cout << "As coordenadas digitadas foram" << endl;
    for (Ponto p:poli.pontos)
        cout << "("<< p.x << " , " << p.y << ") ";
}
```



Residência
em Software



Qual o problema
com esta solução?

```
#include <iostream>
#include <vector>

using namespace std;

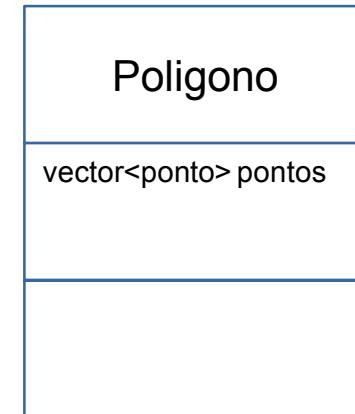
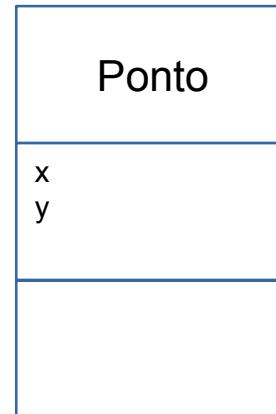
class Ponto {
public:
    float x, y;
};

class Poligono {
public:
    vector<Ponto> pontos;
};

int main () {
    Poligono poli;
    cout << "Criando um polígono!" << endl;
    char sn;
    do {
        cout << "Digite as coordenadas do ponto: ";
        Ponto p;
        cin >> p.x >> p.y;
        poli.pontos.push_back(p);
        cout << "Deseja inserir mais pontos (s/n)?";
        cin >> sn;
    } while (sn!='n');
    cout << "As coordenadas digitadas foram" << endl;
    for (Ponto p:poli.pontos)
        cout << "("<< p.x << " , " << p.y << ") ";
}
```

Sofisticando um pouco mais

- Modifique seu programa
 - Agora ele também deve calcular e mostrar o perímetro do polígono
 - Para isto vai ser necessário usar a distância entre dois pontos
 - Sendo d a distância, x₁, y₁ as coordenadas do primeiro ponto e x₂, y₂ as coordenadas do segundo ponto
 - A distância é dada por



$$d = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$$



Uma solução

```
#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

class Ponto {
public:
    float x, y;
};

class Poligono {
public:
    vector<Ponto> pontos;
};

int main () {
    Poligono poli;
    cout << "Criando um polígono!" << endl;
    char sn;
    do {
        cout << "Digite as coordenadas do ponto: ";
        Ponto p;
        cin >> p.x >> p.y;
        poli.pontos.push_back(p);
        cout << "Deseja inserir mais pontos (s/n)?";
        cin >> sn;
    } while (sn != 'n');
    //Listando os pontos
    cout << "As coordenadas digitadas foram" << endl;
    for (Ponto p:poli.pontos)
        cout << "(" << p.x << " , " << p.y << ")";
    // calculando o perímetro
    float per = 0;
    vector<Ponto>::iterator it2;
    Ponto p1;
    Ponto p2;
    for (auto it = poli.pontos.begin(); it != poli.pontos.end()-1; it++) {
        it2 = it;
        advance(it2,1);
        p1 = *it;
        p2 = *it2;
        per += sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y,2));
    }
    //getting the distance between the final and the initial points
    it2 = poli.pontos.begin();
    p1 = *it2;
    per += sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y,2));
    cout << "Perímetro calculado: " << per << endl;
}
```



Uma solução

```
#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

class Ponto {
public:
    float x, y;
};

class Poligono {
public:
    vector<Ponto> pontos;
};

int main () {
    Poligono poli;
    cout << "Criando um polígono!" << endl;
    char sn;
    do {
        cout << "Digite as coordenadas do ponto: ";
        Ponto p;
        cin >> p.x >> p.y;
        poli.pontos.push_back(p);
        cout << "Deseja inserir mais pontos (s/n)?";
        cin >> sn;
    } while (sn != 'n');
    //Listando os pontos
    cout << "As coordenadas digitadas foram" << endl;
    for (Ponto p:poli.pontos)
        cout << "(" << p.x << " , " << p.y << ")";
    // calculando o perímetro
    // calculando o perímetro
    float per = 0;
    vector<Ponto>::iterator it2;
    Ponto p1;
    Ponto p2;
    for (auto it = poli.pontos.begin(); it != poli.pontos.end()-1; it++) {
        it2 = it;
        advance(it2,1);
        p1 = *it;
        p2 = *it2;
        per += sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y,2));
    }
    //getting the distance between the final and the initial points
    it2 = poli.pontos.begin();
    p1 = *it2;
    per += sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y,2));
    cout << "Perímetro calculado: " << per << endl;
}
```

Qual o problema
com esta solução?



E se quisermos mudar os pontos para (por exemplo) o \mathbb{R}^3 ?

```
#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

class Ponto {
public:
    float x, y;
};

class Poligono {
public:
    vector<Ponto> pontos;
};

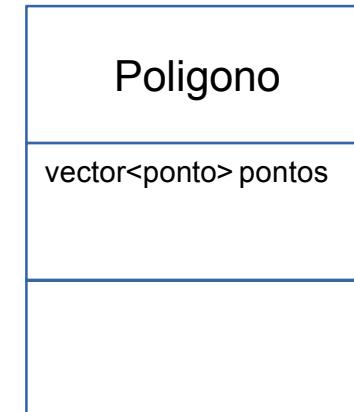
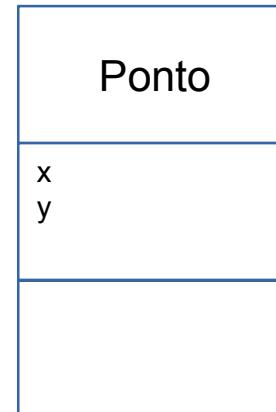
int main () {
    Poligono poli;
    cout << "Criando um polígono!" << endl;
    char sn;
    do {
        cout << "Digite as coordenadas do ponto: ";
        Ponto p;
        cin >> p.x >> p.y;
        poli.pontos.push_back(p);
        cout << "Deseja inserir mais pontos (s/n)?";
        cin >> sn;
    } while (sn != 'n');
    //Listando os pontos
    cout << "As coordenadas digitadas foram" << endl;
    for (Ponto p:poli.pontos)
        cout << "(" << p.x << " , " << p.y << ")" << endl;
    // calculando o perímetro
    float per = 0;
    vector<Ponto>::iterator it2;
    Ponto p1;
    Ponto p2;
    for (auto it = poli.pontos.begin(); it != poli.pontos.end()-1; it++) {
        it2 = it;
        advance(it2,1);
        p1 = *it;
        p2 = *it2;
        per += sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y,2));
    }
    //getting the distance between the final and the initial points
    it2 = poli.pontos.begin();
    p1 = *it2;
    per += sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y,2));
    cout << "Perímetro calculado: " << per << endl;
}
```

Qual o problema
com esta solução?

Repensando as classes

- Métodos

- Funções que permitem que os atributos sejam acessíveis
- Permitem também que as classes cumpram funções (serviços)



Repensando as classes

- Padrão expert
 - (entre outras) é responsabilidade de cada classe prestar o serviço que depende de seus atributos!
- Modifique o seu programa implementando os métodos ao lado!

Ponto
x y
get_x() get_y() le_ponto() escreve_ponto() distancia(Ponto p)

Polígono
vector<ponto> pontos lePontos() listaPontos(); perimetro()

Nova função main

```
int main () {
    Poligono poli;
    //lendo os pontos
    poli.le_pontos();
    //listando os pontos
    poli.lista_pontos();
    // calculando o perímetro
    cout << endl << "Perímetro calculado: " <<
        poli.perimetro() << endl;
}
```

Polígono

vector<ponto> pontos

le_pontos()
lista_pontos();
perimetro()



Residência
em Software

```
class Poligono {
    vector<Ponto> pontos;

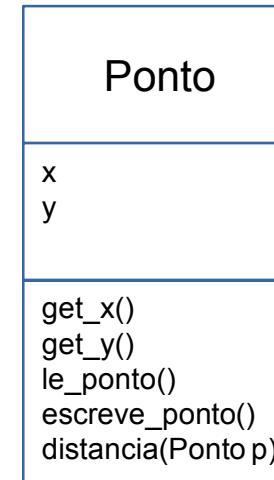
public:
    void lePontos() {
        cout << "Criando um polígono!" << endl;
        char sn;
        do {
            Ponto p;
            p.le_Ponto();
            pontos.push_back(p);
            cout << "Deseja inserir mais pontos (s/n)?" ;
            cin >> sn;
        } while (sn != 'n');
    }

    void listaPontos() {
        cout << "As coordenadas digitadas foram" << endl;
        for (Ponto p: pontos)
            cout << "(" << p.escreve_ponto() << ") ";
    }
}
```

```
float perimetro() {
    float per = 0;
    vector<Ponto>::iterator it2;
    Ponto p1;
    Ponto p2;
    for (auto it = pontos.begin();  
         it != pontos.end()-1; it++) {  
        it2 = it;  
        advance(it2,1);  
        p1 = *it;  
        p2 = *it2;  
        per += p1.distancia(p2);
    }
    //pegando distância entre o primeiro e último
    it2 = pontos.begin();
    p1 = *it2;
    per += p1.distancia(p2);
    return per;
}
```

```
class Ponto {  
    float x, y;  
public:  
  
    float get_x() {  
        return x;  
    }  
  
    float get_y() {  
        return y;  
    }  
  
    void le_Ponto() {  
        cout << "Digite as coordenadas do ponto: ";  
        cin >> x >> y;  
    }  
  
    string escreve_ponto() {  
        return to_string(x) + " , " + to_string(y);  
    }  
  
    float distancia(Ponto p1) {  
        return sqrt( pow(x - p1.get_x(),2) +  
                    pow (y - p1.get_y(),2) );  
    }  
};
```

Classe ponto





- Com esta nova estrutura: modifique a classe ponto para refletir um ponto no \mathbb{R}^3

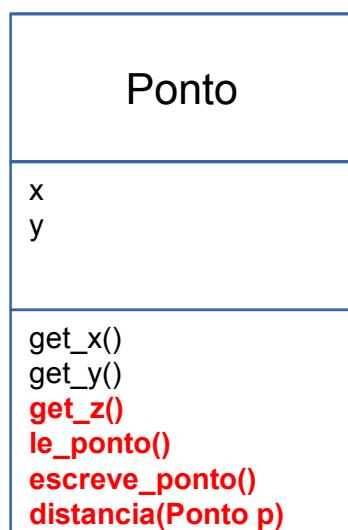
$$d = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2)}$$



Residência
em Software

Onde haverão mudanças?

Onde haverão mudanças?



Onde haverão mudanças?

Ponto

x
y

get_x()
get_y()
get_z()
le_ponto()
escreve_ponto()
distancia(Ponto p)

Polígono

vector<ponto> pontos

lePontos()
listaPontos();
perimetro()

Onde haverão mudanças?

Ponto

x
y

get_x()
get_y()
get_z()
le_ponto()
escreve_ponto()
distancia(Ponto p)

Polígono

vector<ponto> pontos

lePontos()
listaPontos();
perimetro()

Reduzimos o Acoplamento!

Sobrecarga de operadores

- Ainda o mesmo exemplo
 - Suponha que queremos uma maneira de “somar” dois pontos
 - O resultado é a soma das respectivas coordenadas em posições correspondentes
 - Suponha que queremos comparar dois pontos
 - Dois pontos serão iguais se suas coordenadas forem exatamente as mesmas

Solução com métodos

- Como vimos podemos criar um método somar(p)
 - Recebe um ponto p e o soma ao ponto atual
 - Opcionalmente podemos usar um método estático somar(p1, p2)
- Da mesma maneira
 - igual(p) ou igual(p1, p2)

Sobrecarga de operadores

- Uma opção é definir uma sobrecarga
 - Ponto a, b, c; ... c = a+b;
 - c recebe a “soma” entre a e b;
 - Ponto a,b; ... if (a==b) ...
 - Testando se os pontos a e b são iguais

Como fazer

- Definir uma sobrecarga do operador
 - É um método cujo nome começa com a palavra reservada *operator*
 - Seguido do operador desejado
- Ex.
 - **bool operator==(Ponto p) ...**

Considerações

- Apesar de ser usado como um operador, trata-se de um método
 - Ponto a, b, c
 - ...
 - $a = b+c;$
 - É “traduzido” pela invocação de
 - $a = b.operator+(c);$

No nosso exemplo

- Note que tivemos que implementar funções `set_x()` e `set_y()` nos pontos
 - Façam como exercício

```
bool operator==(Ponto p) {
    return p.get_x()==x && p.get_y()==y;
}

Ponto operator+(Ponto p) {
    Ponto p1;
    p1.set_x(p.get_x()+x);
    p1.set_y(p.get_y()+y);
    return p1;
}
```

Outra solução

- (melhor)
- Aqui optamos por implementar um construtor que recebe dois float
 - O que nos obriga a implementar um construtor *default* porque instanciamos de outras formas muitas vezes
 - Fazer como exercício

```
bool operator==(Ponto p) {
    return p.get_x()==x && p.get_y()==y;
}

Ponto operator+(Ponto p) {
    Ponto p1(p.get_x()+x, p.get_y()+y);
    return p1;
}
```

Desafio

- Implementar sobrecarga de operadores para polígonos

- Operador `=` → Cria um polígono com os mesmos pontos de outro
 - `p1 = p2;` // `p1` terá o mesmo conjunto de pontos de `p2`
- Operador `==` → verifica se dois polígonos são iguais
 - Se eles contém exatamente os mesmos pontos, e na mesma ordem*
 - `p1==p2` // será *true* se os pontos de `p1` e `p2` coincidirem (inclusive na mesma posição), ou *false* caso contrário

*Considere que os pontos serão inseridos de forma que SEMPRE formarão um polígono