

Geometria Computacional

Pontos

Prof. Edson Alves

Faculdade UnB Gama

1. Definição de ponto
2. Comparação entre pontos

Definição de ponto

Definição

- Ponto é um termo primitivo da geometria

Definição

- Ponto é um termo primitivo da geometria
- No primeiro livro dos elementos, Euclides define ponto como “*a point is that which has no part*”, que numa tradução livre diz que “ponto é aquilo que não tem parte”

Definição

- Ponto é um termo primitivo da geometria
- No primeiro livro dos elementos, Euclides define ponto como “*a point is that which has no part*”, que numa tradução livre diz que “ponto é aquilo que não tem parte”
- Os pontos são elementos adimensionais, isto é, não tem dimensão alguma

Definição

- Ponto é um termo primitivo da geometria
- No primeiro livro dos elementos, Euclides define ponto como “*a point is that which has no part*”, que numa tradução livre diz que “ponto é aquilo que não tem parte”
- Os pontos são elementos adimensionais, isto é, não tem dimensão alguma
- Em C/C++, pontos podem ser representados como classes ou estruturas, como pares ou como vetores

Definição

- Ponto é um termo primitivo da geometria
- No primeiro livro dos elementos, Euclides define ponto como “*a point is that which has no part*”, que numa tradução livre diz que “ponto é aquilo que não tem parte”
- Os pontos são elementos adimensionais, isto é, não tem dimensão alguma
- Em C/C++, pontos podem ser representados como classes ou estruturas, como pares ou como vetores
- Cada representação possível tem suas vantagens e desvantagens

Representação de pontos como classes

- Representar um ponto utilizando uma classe ou estrutura tem a vantagem da legibilidade

Representação de pontos como classes

- Representar um ponto utilizando uma classe ou estrutura tem a vantagem da legibilidade
- A sintaxe para o uso é mais natural

Representação de pontos como classes

- Representar um ponto utilizando uma classe ou estrutura tem a vantagem da legibilidade
- A sintaxe para o uso é mais natural
- Porém esta representação demanda a implementação dos operadores relacionais para comparações entre pontos

Representação de pontos como classes

- Representar um ponto utilizando uma classe ou estrutura tem a vantagem da legibilidade
- A sintaxe para o uso é mais natural
- Porém esta representação demanda a implementação dos operadores relacionais para comparações entre pontos
- O uso de estruturas simplifica a implementação, uma vez que em competição não há necessidade de encapsulamento dos membros

Representação de pontos como classes

- Representar um ponto utilizando uma classe ou estrutura tem a vantagem da legibilidade
- A sintaxe para o uso é mais natural
- Porém esta representação demanda a implementação dos operadores relacionais para comparações entre pontos
- O uso de estruturas simplifica a implementação, uma vez que em competição não há necessidade de encapsulamento dos membros
- O tipo usado para representar os valores das coordenadas pode ser parametrizado, permitindo o uso da mesma implementação seja com variáveis inteiras, seja com variáveis em ponto flutuante

Exemplo de implementação de ponto usando uma estrutura

```
1 #include <iostream>
2
3 template<typename T>
4 struct Point {
5     T x, y;
6
7     Point(T xv = 0, T yv = 0) : x(xv), y(yv) {}
8 };
9
10 int main() {
11     Point<int> p(1, 2), q;           // Declaração
12     p = q;                           // Atribuição
13
14     if (p == q) {                    // Erro de compilação: o operador == não está definido!
15         p.x = q.y + 1;
16     }
17
18     return 0;
19 }
```

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos
- Usar pares tem como vantagem herdar os operadores de comparação dos tipos escolhidos

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos
- Usar pares tem como vantagem herdar os operadores de comparação dos tipos escolhidos
- A desvantagem é a notação, que utiliza `first` e `second` para acessar as duas coordenadas, ao invés de `x` e `y`, como nas classes e estruturas

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos
- Usar pares tem como vantagem herdar os operadores de comparação dos tipos escolhidos
- A desvantagem é a notação, que utiliza `first` e `second` para acessar as duas coordenadas, ao invés de `x` e `y`, como nas classes e estruturas
- Esta desvantagem pode ser contornada usando as seguintes definições de pré-processador:

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos
- Usar pares tem como vantagem herdar os operadores de comparação dos tipos escolhidos
- A desvantagem é a notação, que utiliza `first` e `second` para acessar as duas coordenadas, ao invés de `x` e `y`, como nas classes e estruturas
- Esta desvantagem pode ser contornada usando as seguintes definições de pré-processador:

```
1 #define x first
2 #define y second
```

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos
- Usar pares tem como vantagem herdar os operadores de comparação dos tipos escolhidos
- A desvantagem é a notação, que utiliza `first` e `second` para acessar as duas coordenadas, ao invés de `x` e `y`, como nas classes e estruturas
- Esta desvantagem pode ser contornada usando as seguintes definições de pré-processador:

```
1 #define x first  
2 #define y second
```

- Esta solução deve ser utilizada com cuidado, pois pode gerar efeitos colaterais indesejados

Representação de pontos como pares e tuplas

- A biblioteca padrão do C++ possui o tipo paramétrico `pair` (par), que pode ser usado para representar pontos
- Usar pares tem como vantagem herdar os operadores de comparação dos tipos escolhidos
- A desvantagem é a notação, que utiliza `first` e `second` para acessar as duas coordenadas, ao invés de `x` e `y`, como nas classes e estruturas
- Esta desvantagem pode ser contornada usando as seguintes definições de pré-processador:

```
1 #define x first
2 #define y second
```

- Esta solução deve ser utilizada com cuidado, pois pode gerar efeitos colaterais indesejados
- Além disso, tuplas (`tuple`) podem ser utilizados diretamente para representar pontos tridimensionais

Exemplo de implementação de ponto usando pares e tuplas

```
1 #include <iostream>
2
3 using Point2D = std::pair<int, int>;           // C++11 em diante
4 using Point3D = std::tuple<int, int, int>;
5
6 #define x first
7 #define y second
8
9 int main()
10 {
11     Point2D p(1, 2), q;                       // Declaração
12     p = q;                                     // Atribuição
13
14     if (p == q) {                             // Ok! Operador == para ints utilizado
15         p.x = q.y + 1;
16     }
17
18     return 0;
19 }
```

Representação de pontos como vetores

- Representar pontos como vetores bidimensionais permite a travessia de conjuntos de pontos em laços por coordenada

Representação de pontos como vetores

- Representar pontos como vetores bidimensionais permite a travessia de conjuntos de pontos em laços por coordenada
- Além disso, é a implementação mais curta para pontos multidimensionais

Representação de pontos como vetores

- Representar pontos como vetores bidimensionais permite a travessia de conjuntos de pontos em laços por coordenada
- Além disso, é a implementação mais curta para pontos multidimensionais
- Porém, a legibilidade fica comprometida, uma vez que as coordenadas são acessadas por índices

Representação de pontos como vetores

- Representar pontos como vetores bidimensionais permite a travessia de conjuntos de pontos em laços por coordenada
- Além disso, é a implementação mais curta para pontos multidimensionais
- Porém, a legibilidade fica comprometida, uma vez que as coordenadas são acessadas por índices
- Esta representação não herda a atribuição e ainda pode gerar confusão com o uso de operadores relacionais

Exemplo de implementação de ponto usando vetores

```
1 #include <iostream>
2
3 using Point = double[2];
4
5 int main()
6 {
7     Point p, q {0, 0};           // Declaração
8     p = q;                       // Erro de compilação: a atribuição não está definida
9
10    if (p < q) {                  // Perigo: comparação entre endereços de ponteiros
11        p[0] = q[1];             // O código compila sem erros
12    }
13
14    return 0;
15 }
```

Comparação entre pontos

- Os operadores relacionais que estarão disponíveis dependem da representação escolhida

Operadores relacionais

- Os operadores relacionais que estarão disponíveis dependem da representação escolhida
- Os operadores fundamentais são a igualdade (operador `==`) e a desigualdade (operador `!=`)

Operadores relacionais

- Os operadores relacionais que estarão disponíveis dependem da representação escolhida
- Os operadores fundamentais são a igualdade (operador `==`) e a desigualdade (operador `!=`)
- Os operadores `<` e `>` também podem ser definidos, embora a semântica destas comparações dependa do contexto e da implementação utilizada

Operadores relacionais

- Os operadores relacionais que estarão disponíveis dependem da representação escolhida
- Os operadores fundamentais são a igualdade (operador `==`) e a desigualdade (operador `!=`)
- Os operadores `<` e `>` também podem ser definidos, embora a semântica destas comparações dependa do contexto e da implementação utilizada
- Mesmo no caso dos pares, que herda a igualdade, é importante implementá-la caso o tipo utilizado para armazenar as coordenadas seja o ponto flutuante, para que seja usado o limiar ϵ

Exemplo de implementação da igualdade

```
1 #include <iostream>
2 #include <cmath>
3
4 const double EPS { 1e-9 };
5
6 template<typename T>
7 bool equals(T a, T b)
8 {
9     if (std::is_floating_point<T>::value)
10         return fabs(a - b) < EPS;
11     else
12         return a == b;
13 }
14
15 template<typename T>
16 struct Point {
17     T x, y;
18
19     Point(T xv = 0, T yv = 0) : x(xv), y(yv) {}
```

Exemplo de implementação da igualdade

```
21  bool operator==(const Point& p) const {
22      return equals(x, p.x) && equals(y, p.y);
23  }
24
25  bool operator!=(const Point& p) const {
26      return not (*this == p);
27  }
28 };
29
30 int main() {
31     Point<double> p(1, 2), q(3*1.0/3, 2);
32
33     if (p == q)
34         p.x = q.y;
35
36     std::cout << "p = (" << p.x << ", " << p.y << ")\n";
37
38     return 0;
39 }
```

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
3. **De BERG**, Mark; **CHEONG**, Otfried. *Computational Geometry: Algorithms and Applications*, 2008.
4. David E. Joyce. *Euclid's Elements*. Acesso em 15/02/2019¹
5. Wikipédia. *Geometria Euclidiana*. Acesso em 15/02/2019².

¹<https://mathcs.clarku.edu/~djoyce/elements/book1/def11.html>

²https://pt.wikipedia.org/wiki/Geometria_euclidiana