Tipos Estruturados

- Até aqui trabalhamos apenas com tipos básicos (disponibilizados pela linguagem), mas para desenvolver programas mais complexos é necessário trabalhar de uma maneira mais abstrata para representar outros tipos de dados.
- Se, por exemplo, deseja-se desenvover um programa que represente pontos em coordenadas cartesianas, o ideal é que seja criado um tipo que agrupe as duas coordenadas (x,y).
- E no caso de se representar o cadastro de alunos matriculados em uma determinada disciplina ? Nesse caso, alguns dados necessários para serem agregados seriam: nome, matrícula, notas, etc.

Tipos Estruturados

- Em C/C++ é possível definir um tipo de dado, cujos campos são compostos de vários tipos básicos da linguagem.
- Para exemplificar considera-se a elaboração de um programa que manipula pontos no espaço R^2:

```
struct ponto
{
  float x;
  float y;
};
```

```
ponto p;   // em C puro: struct ponto p;
```

```
p.x = 10.0;      // atribui 10 ao campo x de p;
p.y =  5.0;      // atribui  5 ao campo y de p;
```

Tipos Estruturados

```
#include <iostream>

using namespace std;

struct ponto
{
  float x;
  float y;
};

int main( )
{
  cout << "Digite as coordenadas do ponto(x,y): ";

  ponto p;
  cin  >> p.x >> p.y;

  cout << "O ponto fornecido foi: (" << p.x << "," << p.y << ")" << endl;

  return 1;
}
```

Ponteiros para Estruturas

- Da mesma forma que é possível declarar variáveis do tipo de uma estrutura, também é possível declarar variáveis do tipo ponteiro para uma estrutura:

```
ponto p;
```

```
ponto *pp;
```

- Se a variável *pp* armazenar o endereço de uma estrutura, pode-se acessar os campos dessa estrutura indiretamente, por meio de seu ponteiro:

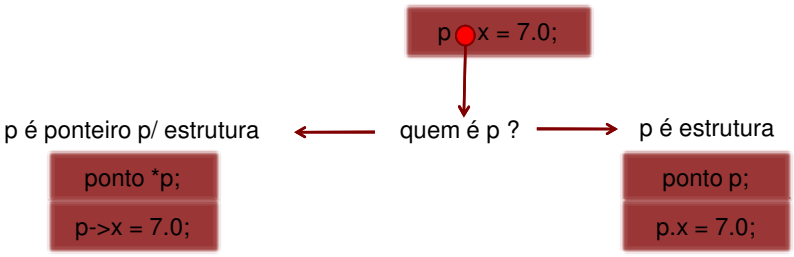
```
(*pp).x = 12.0;
```

- A linguagem oferece outro operador ***(->)*** para fazer esse acesso:

```
pp->x = 12.0;
```

Ponteiros para Estruturas

- Uma dúvida comum para programadores iniciantes em C/C++ consiste em saber quando usar o *ponto* e quando usar a *seta* para acessar um campo de uma variável estrutura:



Referências para Estruturas

- É possível também criar variáveis do tipo de uma estrutura como referência de uma outra variável do mesmo tipo.
- A notação ***X&*** representa referência para ***X***, onde ***X*** é um tipo (básico ou não).
- O principal uso de referências é para passagem de parâmetros e retorno de funções.

Referências

- A forma de utilização também é similar ao uso para variáveis que são referência para tipos básicos da linguagem, e da mesma forma, também devem obrigatoriamente ser inicializadas na declaração:

```
ponto p = {3.0,4.0};        // p é variável do tipo ponto
ponto& rp = p;              // rp é referência para p
ponto q = rp;               // q = (3.0,4.0)
q.x = 7.0;                  // p = (7.0,3.0)
```

- O acesso a campos de referências para estruturas é feito através do operador *ponto (.)*.

Passagem de Estruturas p/ Funções

- Para exemplificar a passagem de estruturas como argumento de funções, pode-se utilizar o exemplo da captura de pontos do R^2.
- Inicialmente cria-se duas funções: uma responsável pela leitura e outra pela impressão dos pontos:

```
void imprime( ponto  p );
void captura( ponto *p );
```

Passagem de Estruturas p/ Funções

```
#include <iostream>

using namespace std;

struct ponto
{
  float x;
  float y;
};

void captura( ponto *p );
void imprime( ponto  p );

int main( )
{
  ponto p;

  captura( &p );

  imprime( p );

  return 1;
}
```

```
void captura( ponto *p )
{
  cout << "Digite as coordenadas do ponto(x,y): ";
  cin  >> p->x >> p->y;
}

void imprime( ponto p )
{
  cout << "O ponto fornecido foi: (" << p.x << "," << p.y << ")" << endl;
}
```

- Por qual motivo foi necessário definir a função captura recebendo como argumento o ponteiro da estrutura ponto ?
- E se fosse utilizado um argumento do tipo ***ponto&*** ?

Passagem de Estruturas p/ Funções

```
#include <iostream>

using namespace std;

struct ponto
{
  float x;
  float y;
};

void captura( ponto& p );
void imprime( ponto  p );

int main( )
{
  ponto p;

  captura( p );

  imprime( p );

  return 1;
}
```

```
void captura( ponto& p )
{
  cout << "Digite as coordenadas do ponto(x,y): ";
  cin  >> p.x >> p.y;
}

void imprime( ponto p )
{
  cout << "O ponto fornecido foi: (" << p.x << "," << p.y << ")" << endl;
}
```

Alocação Dinâmica de Estruturas

- Assim como os vetores, as estruturas também podem ser alocadas dinamicamente:

```
double *r;
r = new double;
…
*r = 1.2;
…
delete r;
```

```
ponto *p;
p = new ponto;
…
p->x = 12.0;
…
delete p;
```

- A diferença, em termos de sintaxe, é que para alocar memória para uma única variável, não se utiliza o ***operador []***.
- É possível alocar memória para uma variável de um tipo básico da linguagem também.

Estruturas Agregadas

- Os campos de uma estrutura podem conter variáveis de tipos de outras estruturas:

```
struct circulo
{
  float x, y;
  float r;
};
```

```
struct ponto
{
  float x;
  float y;
};
```

```
struct circulo
{
  ponto c;
  float r;
};
```

Estruturas Agregadas

- Como exemplo, apresenta-se uma função que calcula a distância entre dois pontos.
- Utilizando-se dessa função, é possível implementar uma função que receba como parâmetro um círculo e um ponto e informe se o ponto está dentro do círculo:

```
float distancia ( ponto* p, ponto* q )
{
  float x2 = (q->x-p->x)*(q->x-p->x);
  float y2 = (q->y-p->y)*(q->y-p->y);

  return sqrt( x2+y2 );
}
```

```
bool interior ( circulo* c, ponto* p )
{
  float d = distancia( &c->p, p );

  return (d < c->r);
}
```

- Com essas funções e as estruturas apresentadas é possível escrever um programa que leia pontos do R^2 e informe se esses estão dentro de um círculo também lido, por exemplo.

Vetores de Estruturas

- É possível também o uso de vetores para agrupar estruturas, ou seja, vetores cujos elementos são estruturas.
- É possível portanto armazenar um conjunto de variáveis do tipo ponto em um vetor:

```
// Alocado Estaticamente:
ponto vp[10];
…
vp[0].x = 0.0;
vp[0].y = 0.0;
…
vp[9].x = 3.2;
vp[9].y = 5.4;
…
```

```
// Alocado Dinamicamente:
ponto *vp = new[10];
…
vp[0].x = 0.0;
vp[0].y = 0.0;
…
vp[9].x = 3.2;
vp[9].y = 5.4;
delete [] vp;
```

Vetores de Estruturas

- É possível também ter um vetor de estruturas agregado a um campo de uma outra estrutura.
- Como exemplo apresenta-se a estrutura polígono abaixo:

```
struct ponto
{
  float x;
  float y;
};
```

```
struct poligono
{
  int n;          // número de pontos do polígono
  ponto* vp;      // vetor com os pontos do polígono
};
```

Vetores de Estruturas

- Com isso é possível criar-se uma funções que construam, destruam e imprimam variáveis do tipo polígono:

```
poligono *criaPol( int n, ponto *vp )
{
  poligono* pol = new poligono;

  pol->n  = n;
  pol->vp = new ponto[n];

  for( int i=0; i<n; i++ )
    pol->vp[i] = vp[i];

  return pol;
}
```

```
void destroiPol( poligono* pol )
{
  delete [] pol->vp;
  delete pol;
}
```

```
void imprimePol( poligono *pol )
{
  cout << "Número de Vértices do Polígono: " << pol->n << endl;

  for( int i=0; i< pol->n; i++ )
  {
    float x = pol->vp[i].x;
    float y = pol->vp[i].y;

    cout << "ponto " << i << ": (" << x << "," << y << ")" << endl;
  }
}
```

Vetores de Estruturas

```
#include <iostream>

using namespace std;

struct ponto
{
  float x;
  float y;
};

struct poligono
{
  int    n;
  ponto *vp;
};

poligono  *criaPol( int n, ponto *vp );
void       destroiPol( poligono *pol );
void       imprimePol( poligono *pol );
```

```
int main( )
{
  ponto vp[3];

  vp[0].x = 0.0;
  vp[0].y = 0.0;

  vp[1].x = 0.0;
  vp[1].y = 1.0;

  vp[2].x = 1.0;
  vp[2].y = 0.0;

  poligono *pol = criaPol( 3, vp );

  imprimePol( pol );

  destroiPol( pol );

  return 1;
}
```

Enumerados

- Uma enumeração é uma constante de valores inteiros com nomes que especifica valores legais possíveis para uma variável daquele tipo;
- É uma forma mais elegante de organizar valores constantes

```
enum Inteiro
{
  ZERO,
  UM,
  DOIS,
  TRES
};
```

```
enum TipoPol
{
  TRIANGULO      =   3,
  QUADRILATERO   =   4,
  PENTAGONO      =   5,
  HEXAGONO       =   6,
  HEPTAGONO      =   7
};
```

```
// Protótipo da função:
Poligono *criaPol (TipoPol n, ponto *vp);
```

```
// Chamada da função:
pol = criaPol (TRIANGULO, vp);
```