

INF 112 - Programação II

Tipos Abstratos de Dados

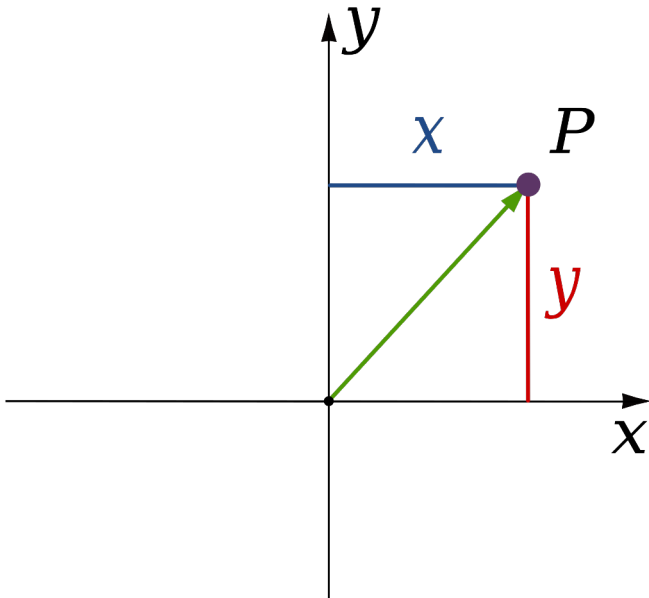
Revisando Structs

Structs

- Em C/C++ podemos criar novos tipos
- Úteis para representar conceitos mais complexos

Struct Ponto

apenas x e y



```
struct ponto_t {  
    double x;  
    double y;  
};
```

structs em C++

- Um pouco mais simples do que em C
- Por enquanto, não precisamos de typedef

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

structs em C++

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    → ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

nome	end(&)	val(*)
main	0x0048	
	0x0044	
	0x0040	
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

structs em C++


```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    → ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	??
ponto_a.y	0x0040	??
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

structs em C++

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
     ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	7
ponto_a.y	0x0040	??
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

structs em C++

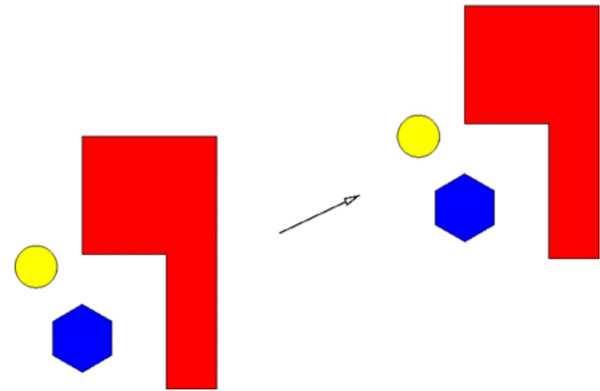
```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    → std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	7
ponto_a.y	0x0040	9
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

Passagem por referência de structs

- Vamos implementar um procedimento de translação



```
void translacao(ponto_t &ponto, float dx, float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```

#include <iostream>

struct ponto_t {
    float x;
    float y;
};

void translacao(ponto_t &ponto, float dx,
               float dy) {
    ponto.x += dx;
    ponto.y += dy;
}

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}

```

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	7
ponto_a.y	0x0040	9
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

```

#include <iostream>

struct ponto_t {
    float x;
    float y;
};

void translacao(ponto_t &ponto, float dx,
               float dy) {
    ponto.x += dx;
    ponto.y += dy;
}

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}

```

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	7
ponto_a.y	0x0040	9
transl...	0x003c	
&ponto	0x0038	0x0044
dx	0x0034	3
dy	0x0030	1
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

```
#include <iostream>
```

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

```
void translacao(ponto_t &ponto, float dx,  
               float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```
int main() {  
    ponto_t ponto_a;  
    ponto_a.x = 7;  
    ponto_a.y = 9;  
    translacao(ponto_a, 3, 1);  
    std::cout << ponto_a.x << std::endl;  
    std::cout << ponto_a.y << std::endl;  
    return 0;  
}
```

Referência

Observe que podemos usar .

Sem &. Referência

nome

main

ponto_a.x

ponto_a.y

transl...

&ponto

dx

dy

end(&)

0x0048

0x0044

0x0040

0x003c

0x0038

0x0034

0x0030

0x002c

0x0028

0x0024

0x0020

0x001c

0x0018

0x0014

0x0010

0x000c

0x0008

0x0004

0x0000

val(*)

10

10

0x0044

3

1

```

#include <iostream>

struct ponto_t {
    float x;
    float y;
};

void translacao(ponto_t &ponto, float dx,
               float dy) {
    ponto.x += dx;
    ponto.y += dy;
}

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}

```

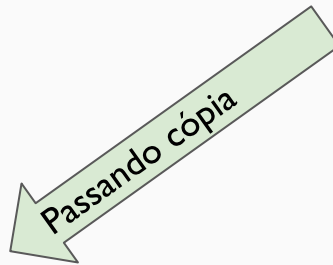
nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	10
ponto_a.y	0x0040	10
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

Qual o problema com esta chamada?

```
void translacao(ponto_t ponto, float dx, float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```
#include <iostream>
```

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

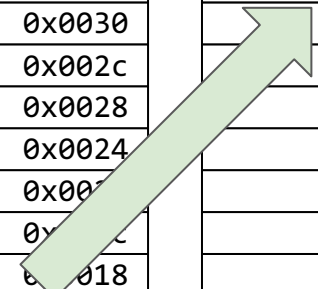


```
void translacao(ponto_t ponto, float dx,
                float dy) {
```

```
    ponto.x += dx;
    ponto.y += dy;
}
```

```
int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	7
ponto_a.y	0x0040	9
transl...	0x003c	
ponto.x	0x0038	7
ponto.y	0x0034	9
dx	0x0030	3
dy	0x002c	1
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	




```
#include <iostream>
```

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

```
void translacao(ponto_t ponto, float dx,
               float dy) {
```

```
    ponto.x += dx;
    ponto.y += dy;
```

```
}
```

```
int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

Passando cópia

nome	end(&)	val(*)
main	0x0048	
ponto_a.x	0x0044	7
ponto_a.y	0x0040	9
transl...	0x003c	
ponto.x	0x0038	10
ponto.y	0x0034	10
dx	0x0030	3
dy	0x002c	1
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

Alocando structs no heap

- Fazemos uso de new e delete
- Similar a uma variável
- Uso comum para implementar listas etc.

```
ponto_t *p = new ponto_t;  
delete p;
```

Tipos Abstrato de Dados (TAD)

TADs e Operações

- Quais operações uma coleção suporta?
- Pense em um conjunto matemático

TADs e Operações

- Quais operações uma coleção suporta?
- Pense em um conjunto matemático
 - adicionar (união) elemento
 - remover (complemento) elemento
 - interseção
 - número de elementos
 - domínio
 - . . .

Tipos Abstratos de Dados (TADs)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
 - conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- A implementação do algoritmo em uma linguagem de programação exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.

Contrato

- TADs são contratos
- Funções que operam em cima da memória

Encapsulamento

- Conceito importante em TADs
- Usuário:
 - Enxerga a interface
 - Não se preocupa, em primeiro momento, como é o TAD por baixo

Então TADs são structs?

TADs vs Structs

- **Não!**
- TADs são um conceito mais geral
 - Existem em qualquer tipo de linguagem
- Em C++
 - Sim, mapeiam bem para structs/classes + .h
 - Ou para interfaces
 - Assuntos futuro

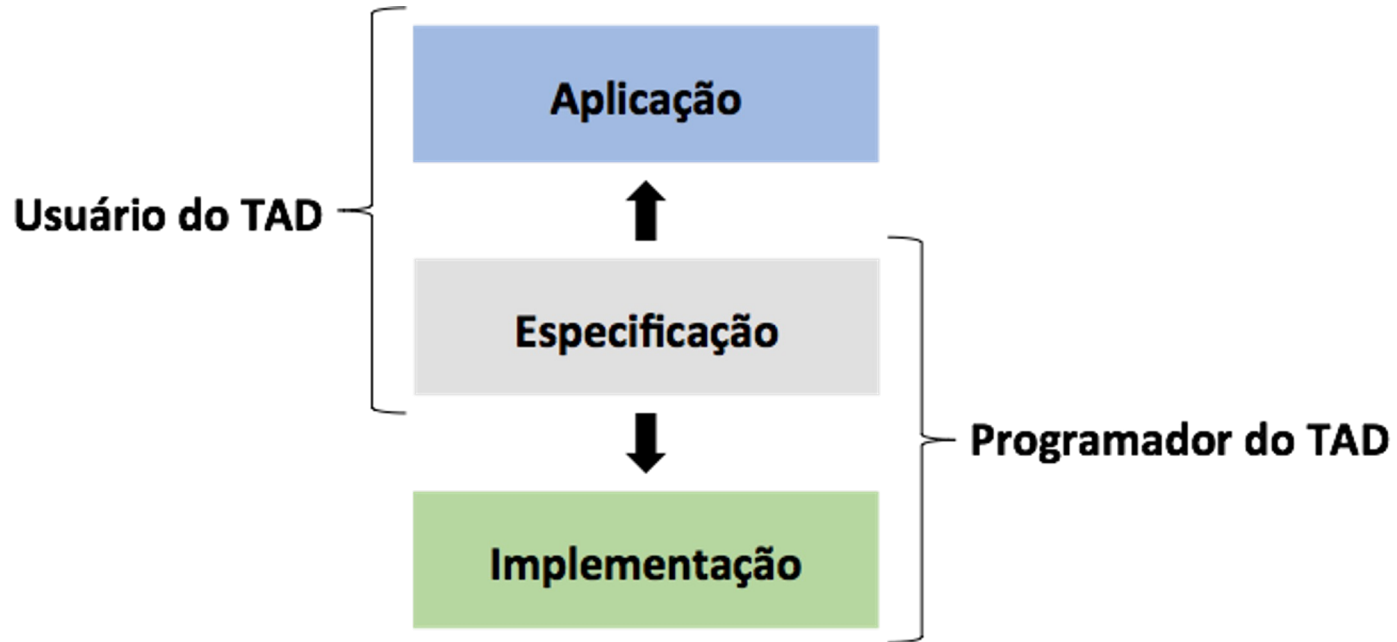
TADs vs Algoritmos

- Algoritmo
 - Sequência de ações executáveis
entrada → saída
 - Exemplo: “Receita de Bolo”
 - Algoritmos usam TADs
- TADs
 - Contrato
+
 - Memória

Tipos Abstratos de Dados (TADs)

- Podemos considerar TADs como generalizações de tipos primitivos e procedimentos como generalizações de operações primitivas.
- O TAD encapsula tipos de dados. A definição do tipo e todas as operações ficam localizadas numa seção do programa.
 - Os usuários do TAD só tem acesso a algumas operações disponibilizadas sobre esses dados

Tipos Abstratos de Dados (TADs)



Tipos Abstratos de Dados (TADs)

- TADs são um conceito de programação
- Vamos aprender como implementar os mesmos usando classes e objetos
- Outras linguagens
 - structs + funções (C)
 - traits (Rust)
 - duck typing (Python, Ruby)
 - **classes e interfaces (Java, C++)**

Perguntas TAD

Supondo que vamos criar um TAD qualquer

1. Como organizar a memória?

2. Quais operações?

a. Assinaturas

b. Contratos

A primeira pergunta é mais de implementação, o TAD é descrito pela 2.

Como fazer um TAD ponto?

Primeiro problema

- Quais dados temos que representar?

Primeiro problema

- Quais dados temos que representar?
 - Valor no eixo-x
 - Valor no eixo-y

Em Código

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
}
```

Alguns pontos importantes:

1. O uso de `_` é estilo, facilita a codificação
2. Vamos chamar de **P**onto por estilo também. Isto é, iniciando com maiúsculo

Até agora temos apenas um struct

Diferenças de C

1. Não precisamos de **typedef**
2. Podemos associar métodos ao struct

Segundo Problema

- Quais problemas?
 - Construir o ponto
 - Translação
 - Rotação
 - Imprimir

Imprimir o ponto

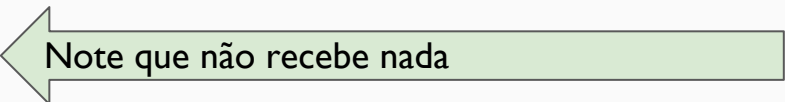
Operação mais simples

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout << "x= " << _x_val << "y=" << _y_val << std::endl;  
    }  
};
```

Imprimir o ponto

Operação mais simples (omitindo imports)

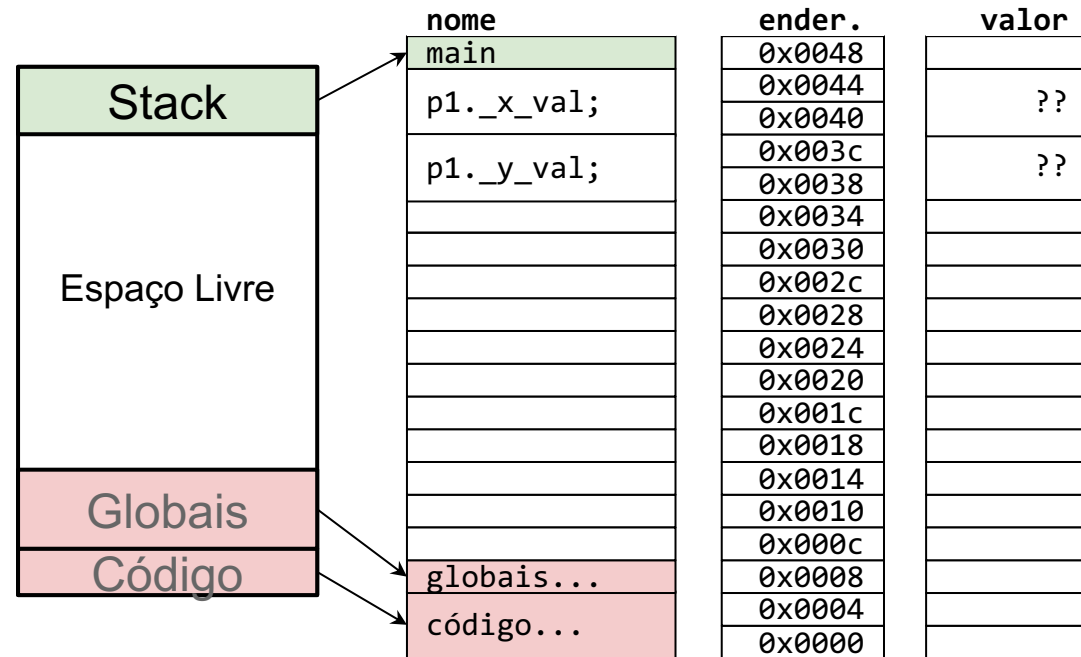
```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout << "x= " << _x_val << "y=" << _y_val << std::endl;  
    }  
};
```



Note que não recebe nada

Na memória

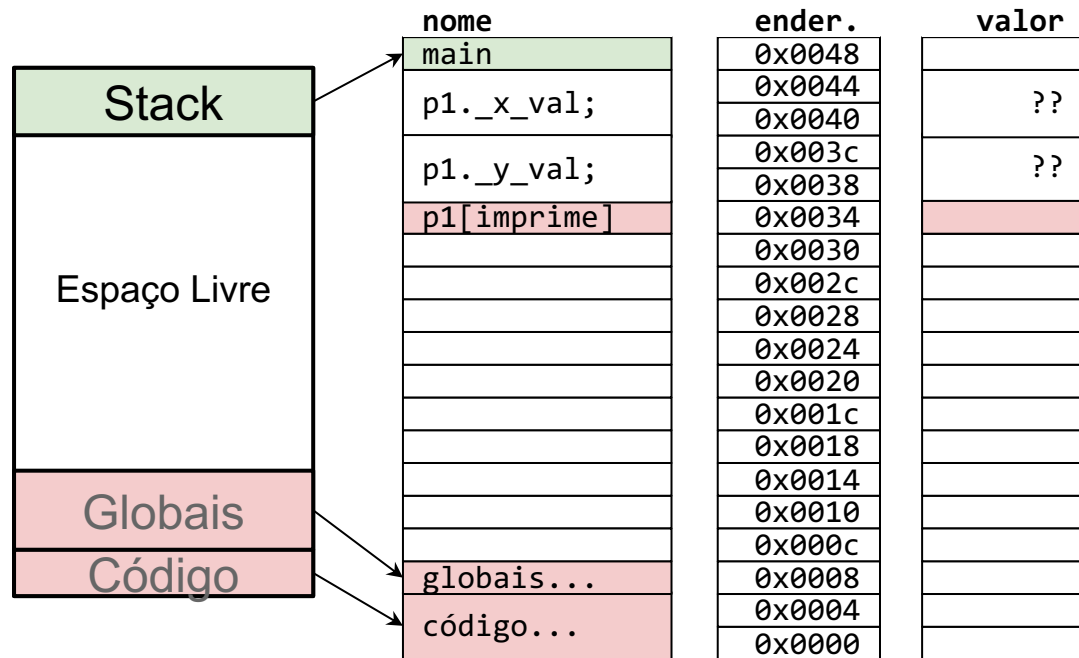
```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        // ...;  
    }  
};  
  
int main() {  
→ Ponto p1;  
}
```



Na memória (muito simplificada!!)

A máquina “guarda” que existe uma operação imprime no struct

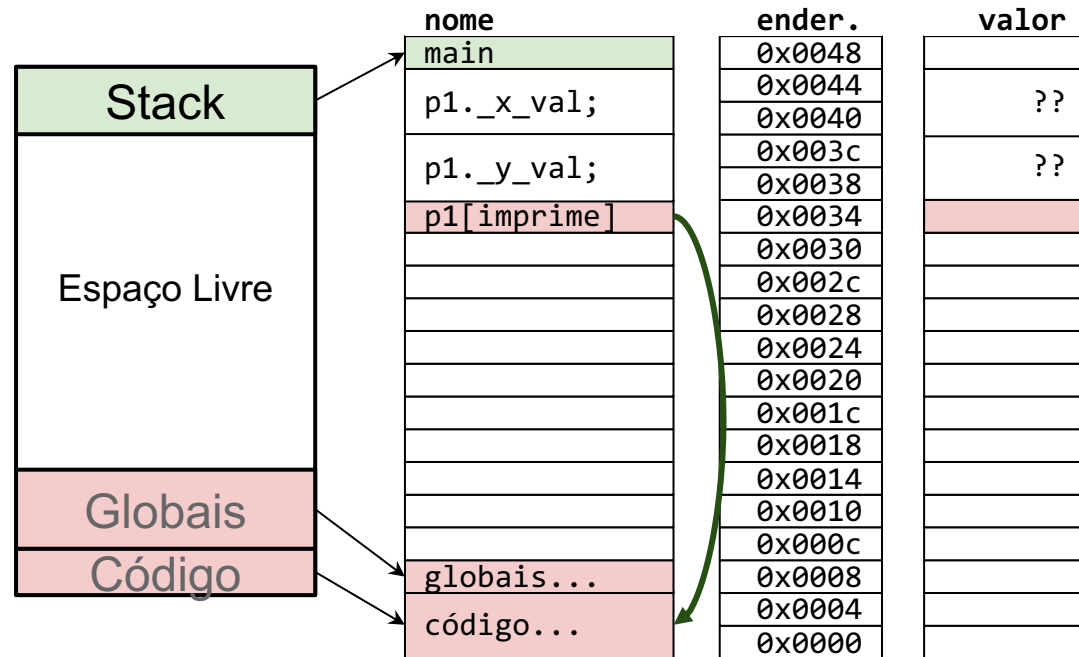
```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        // ...;  
    }  
};  
  
int main() {  
    Ponto p1;  
}
```



Na memória (muito simplificada!!)

A mesma sabe exatamente qual código chamar

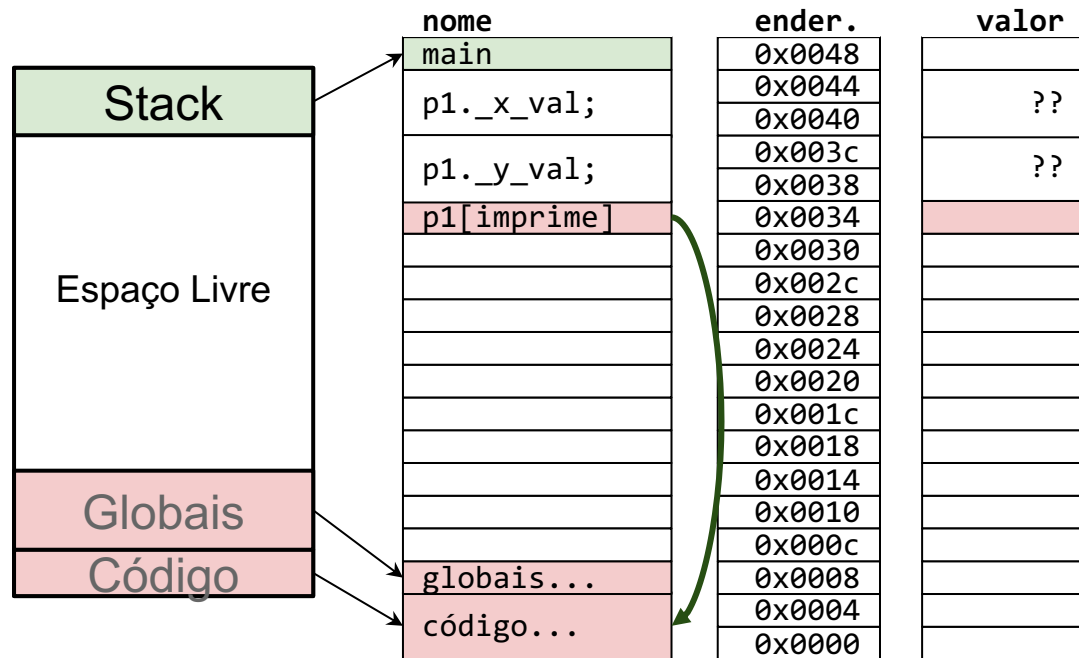
```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        // ...;  
    }  
};  
  
int main() {  
    Ponto p1;  
}
```



Na memória (muito simplificada!!)

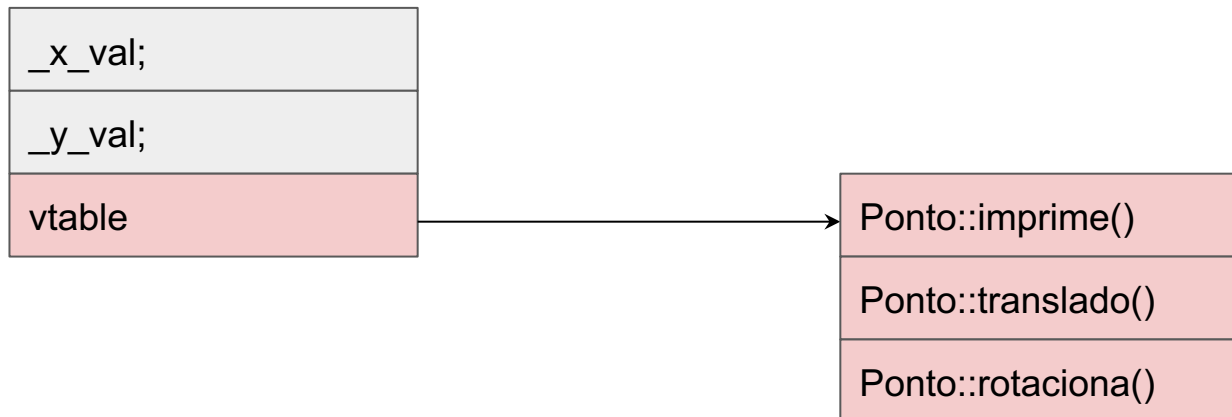
Se você conhece ponteiros para funções, mesma ideia.

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        // ...;  
    }  
};  
  
int main() {  
    Ponto p1;  
}
```



A Tabela de Funções (vtable)

- Guarda funções que pertencem ao struct
- Como funciona uma chamada?



Chamada de métodos

- Vamos usar o nome de método para uma função ou procedimento atrelado ao struct
- Facilita nossa transição para classes
- Vamos também simplificar o desenho da memória

Chamada de métodos

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout <<  
_x_val;  
    }  
};  
  
int main() {  
    Ponto p1;  
    p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
}
```


	_x_val	_y_val
main::p1 (stack)	??	??

Chamada de métodos

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout <<  
_x_val;  
    }  
};  
  
int main() {  
    Ponto p1;  
    p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
}
```

	_x_val	_y_val
main::p1 (stack)	??	??

Chamada de métodos

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout <<  
_x_val;  
    }  
};  
  
int main() {  
    Ponto p1;  
     p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
}
```

	_x_val	_y_val
main::p1 (stack)	2	??

Chamada de métodos

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout <<  
_x_val;  
    }  
};  
  
int main() {  
    Ponto p1;  
    p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
}
```

	_x_val	_y_val
main::p1 (stack)	2	3

Qual o valor de `_x_val`?

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout <<  
        _x_val;  
    }  
};  
  
int main() {  
    Ponto p1;  
    p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
}
```

	<code>_x_val</code>	<code>_y_val</code>
main::p1 (stack)	2	3

Qual o valor de `_x_val`? `_x_val=2;`

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    void imprime() {  
        std::cout <<  
_x_val;  
    }  
};  
  
int main() {  
    Ponto p1;  
    p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
}
```

	<code>_x_val</code>	<code>_y_val</code>
main::p1 (stack)	2	3

Chamada de métodos

- Ao chamar o método usamos a memória alocada naquele struct
- Eventualmente vamos chamar isto de objeto

TADs e Estado

Em alto nível, é importante:

1. Lembrar que TADs são dados + operações
2. Dados são mutáveis, ou seja, guardam estado!
3. Aqui estamos fazendo isto com um struct.

E agora?

```
int main() {  
    Ponto p1;  
    p1._x_val = 2;  
    p1._y_val = 3;  
    p1.imprime();  
    Ponto p2 = new Ponto;  
    p2->_x_val = 9;  
    p2->_y_val = 9;  
    p2->imprime();  
}
```

	_x_val	_y_val
main::p1 (stack)	2	3
main::p2 (heap)	9	9

Chamada de métodos

- Cada chamada opera na memória que foi alocada
- Não importa se foi no stack ou no heap
- Em outras palavras, opera no estado

Construtores

- Funções que iniciam o struct
- Chamadas de construtores

Construtor

- Note que o mesmo não tem um valor de retorno

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

Construtor


- Note que o mesmo não tem um valor de retorno

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	??	??

Construtor

- Note que o mesmo não tem um valor de retorno

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	??	??

Construtor

- Note que o mesmo não tem um valor de retorno

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	2	??

Construtor

- Note que o mesmo não tem um valor de retorno

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	2	1

Construtor

- Podemos chamar de duas formas possíveis

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	2	1

Construtor

- Podemos chamar de duas formas possíveis

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	2	1
main::p2 (stack)	??	??

Construtor

- Podemos chamar de duas formas possíveis


```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```

	_x_val	_y_val
main::p1 (stack)	2	1
main::p2 (stack)	2	??

Construtor

- Podemos chamar de duas formas possíveis

```
struct Ponto {  
    // Dados  
    double _x_val;  
    double _y_val;  
  
    // Operacoes  
    // 1. Construtor  
    Ponto(double x_val, double y_val) {  
        _x_val = x_val;  
        _y_val = y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
}
```



	_x_val	_y_val
main::p1 (stack)	2	1
main::p2 (stack)	2	3

Pensando em um estado

- Temos que iniciar o estado
 - Construtor
- Depois
 - Podemos ler e escrever do mesmo

Código Translação

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
        _x_val += outro._x_val;  
        _y_val += outro._y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```

```
$ ./programa  
x=2 y=1
```

	_x_val	_y_val
main::p1	2	1
main::p2	2	3


Note que o método opera em p1 e p2

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
        _x_val += outro._x_val;  
        _y_val += outro._y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```

```
$ ./programa  
x=2 y=1
```

	_x_val	_y_val
main::p1	2	1
main::p2	2	3


Note que o método opera em p1 e p2

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
         _x_val += outro._x_val;  
        _y_val += outro._y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```

```
$ ./programa  
x=2 y=1
```

	_x_val	_y_val
main::p1	2	1
main::p2	2	3

Note que o método opera em p1 e p2

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
        _x_val += outro._x_val;  
         _y_val += outro._y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```

```
$ ./programa  
x=2 y=1
```

	_x_val	_y_val
main::p1	4	1
main::p2	2	3

Note que o método opera em p1 e p2

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
        _x_val += outro._x_val;  
        _y_val += outro._y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```

```
$ ./programa  
x=2 y=1
```

	_x_val	_y_val
main::p1	4	4
main::p2	2	3

Note que o método opera em p1 e p2

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
        _x_val += outro._x_val;  
        _y_val += outro._y_val;  
    }  
};  
  
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```


```
$ ./programa  
x=2 y=1  
x=4 y=4
```

	_x_val	_y_val
main::p1	4	4
main::p2	2	3

Note que o método opera em p1 e p2

```
struct Ponto {  
    // .. ocultando o construtor e o imprime  
    // Operacoes  
    void translado(Ponto &outro) {  
        _x_val += outro._x_val;  
        _y_val += outro._y_val;  
    }  
};
```

```
int main() {  
    Ponto p1 = Ponto(2, 1);  
    Ponto p2(2, 3);  
    p1.imprime();  
    p1.translado(p2);  
    p1.imprime();  
    p2.imprime();  
}
```



```
$ ./programa  
x=2 y=1  
x=4 y=4  
x=2 y=3
```

	_x_val	_y_val
main::p1	4	4
main::p2	2	3

Notas Finais

- Estamos iniciando o assunto da matéria
- Lembre-se de:
 - TADs \rightarrow dados + operações
 - TADs guardam estado
 - Podemos usar um struct
 - Meio não definição

Próxima Aula

- Criando TADs mais complexos
- Separando em módulos

Aonde queremos chegar

Com TADs queremos que o resto do programa seja cliente. Apenas use as operações do mesmo.

