Estrutura de Dados

Profa. Marta

Aula1: TAD – Tipos Abstrato de Dados

TAD – Tipo Abastrato de Dados

- Quando escrevemos um algoritmo, abstraimos um problema do mundo real e transformamos basicamente em DADOS e OPERAÇÕES. Por exemplo:

Mundo real | **Problema** | **Solução**

Um programa para calcular a idade de uma pessoa.

Operação: Calcular
Dados: Data Nascimento

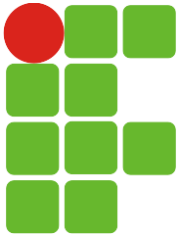
Um programa para registrar pessoas em uma fila.

Solução

Operação:
1)RegistrarSaida
2)RegistrarEntrada
Dados: NomePessoa

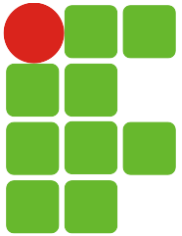
Um programa para cadastrar dados escolares.

TAD



- Da forma que estamos trabalhando conseguimos reutilizar nossas operações em outros problemas ou soluções?
- Conseguimos ocultar regras importantes de implementação?
- A alteração das operações impacta em todo programa?
- O que fazer??

TAD – Tipo Abastrato de Dados



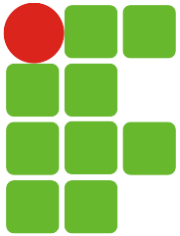
- TAD – É a base para toda a disciplina!

- TAD é uma técnica de programação na qual você especifica os dados e quais operações realizará sobre dados.

Exemplo 1:

```
#include <stdio.h>
File* f;
f=fopen("c:\\test.txt", "r");
fclose(f);
fgetc
fputc
```

TAD



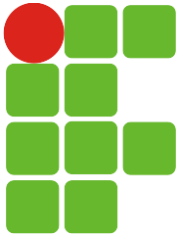
- Exemplo 2: Um programa que realize operações com uma string:

```
#include <stdio.h>
#include <string.h>
int main(){
   char str[100],str1[100],str2[100];
   printf("Entre com uma sequencia de caracteres:");
   gets(str);
   strcpy(str,str1);
   strcat(str,str2);
   printf("tamanho:%d",strlen(str));
   return 0;
}
```

strcpy, strcat, strlen -> São funções pertencentes ao arquivo string.h

String.h -> Agrupa tipo de dado string e conjunto de operações.

TAD

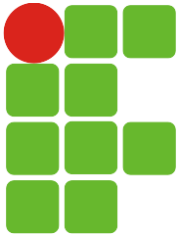


- Exemplo: Se em várias partes do programa precisamos ler um arquivo. Então criamos ou usamos um TAD com operações e um tipo (arquivo).

Entrada: arquivo→ Operação → **Saída**: Exemplo: abertura do arquivo

Porque é boa prática usar TAD?

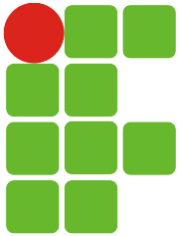
TAD – Vantagens:



1. **Encapsulamento**
2. **Segurança -** o usuário não tem acesso direto aos dados
3. **Flexibilidade** – Altera o TAD e não precisa altera as aplicação que usam o TAD
4. **Reutilização**
5. **Ocultação da informação –** os dados são alterados apenas por meio das operações

COMO CRIAR O PRÓPRIO TAD?

TAD – Exemplo 1



- Considere a criação de um tipo de dados para representar um ponto R^2 .

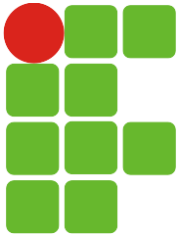
Crie as seguintes operações com esse ponto geométrico:

- Cria – Cria um ponto com as coordenadas x e y.
- Libera – Liberar memória alocado por um ponto.
- Acessa – Devolve as coordenadas de um ponto.
- Atribui – Atribuir novos valores a x e y.
- Distância – Calcula a distância entre dois pontos.

PRIMEIRO: ABSTRAIR OS DADOS?

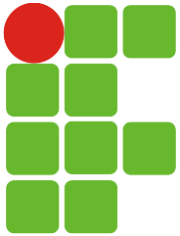
SEGUNDO: QUAIS OPERAÇÕES?

TAD



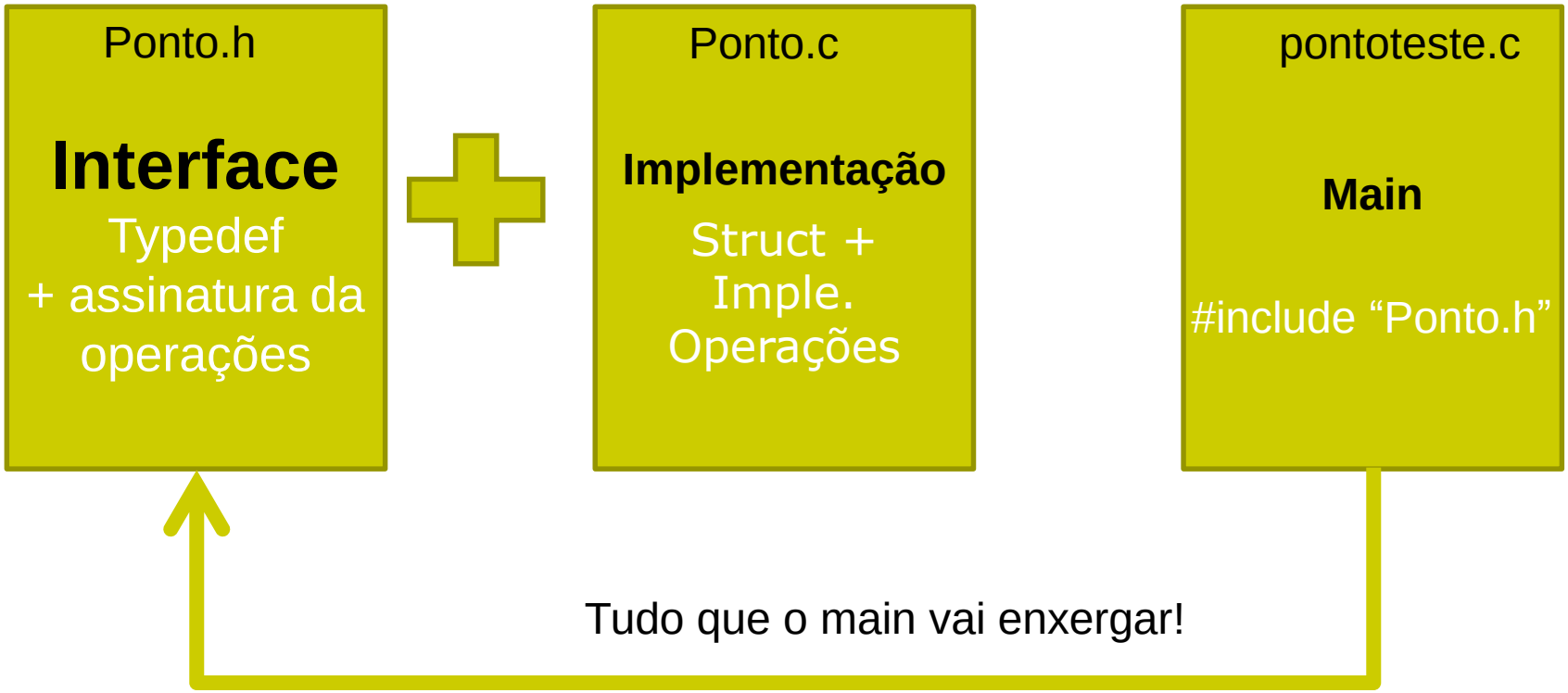
```
/*definir a estrutura do ponto*/
struct ponto{
        float x;
        float y;
};
typedef struct ponto Ponto;
```

TAD



```
Ponto* cria(float x, float y);

/*Libera memoria de um ponto criado*/
void libera(Ponto* p);

/*Devolve coordenadas*/
void acessa(Ponto* p, float* x, float* y);

/*atribui coordenadas x, y*/
void atribui(Ponto* p, float x, float y);

float distancia(Ponto* p1, Ponto* p2);
```

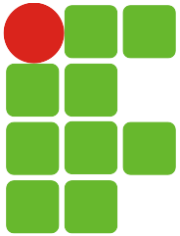
TAD na linguagem C – arquivos:



Onde eu coloco os dados e a operações?
Onde coloco struct e o typedef?
Quero distribuir minha biblioteca e ocultar a implementação?
POSSO ACESSAR A ESTRUTURA PONTO.X NO MAIN?

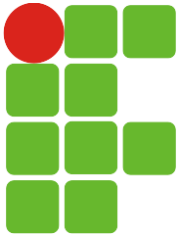
Ponto.h

```
/*TAD ponto(x,y)*/
typedef struct ponto Ponto;
```



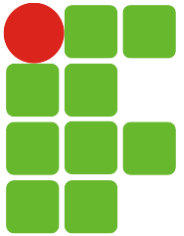
```
/*Criar coordenadas x,y*/
Ponto* cria(float x, float y);

/*Libera memoria de um ponto criado*/
void libera(Ponto* p);

/*Devolve coordenadas*/
void acessa(Ponto* p, float* x, float* y);

/*atribui coordenadas x, y*/
void atribui(Ponto* p, float x, float y);

/*Calcula distância*/
float distancia(Ponto* p1, Ponto* p2);
```

Ponto.c



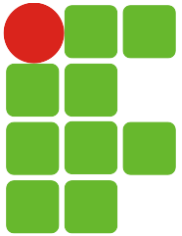
```
#include "Ponto.h"



/*definir a estrutura do ponto*/
struct ponto{
        float x;
        float y;
};
```

Ponto.c



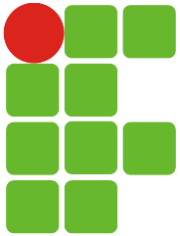
```
//Criar dinamicamente um ponto e inicializar os campos
Ponto* cria(float x, float y)
{
        Ponto* p = (Ponto*) malloc(sizeof(Ponto));
        if (p== NULL){
                printf("Memoria insuficiente.");
                exit(1);
        }

        p->x=x;
        p->y=y;
        return p;

}
```

Ponto.c



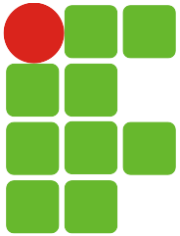
```
void acessa(Ponto* p, float* x, float* y){
        *x = p->x;
        *y = p->y;

}
void atribui(Ponto* p, float x, float y){
        p->x = x;
        p->y = y;
}

float distancia(Ponto* p1, Ponto* p2)
{
        float dx = p2->x - p1->x;
        float dy = p2->y - p1->y;
```

Ponto.c



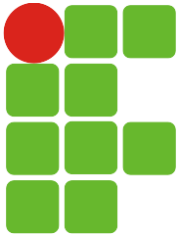
```
float distancia(Ponto* p1, Ponto* p2)
{
        float dx = p2->x - p1->x;
        float dy = p2->y - p1->y;

        return sqrt(dx * dx + dy *dy);
}
```

Pontoteste.c



```
#include "Ponto.h"
int main(){
        printf("\n Criando ponto");
        Ponto* p = (Ponto*) cria(10.00,20.00);
        Ponto* p1 = (Ponto*) cria(50.00,60.00);
        float x1,y1;
        atribui(p,30.00,40);
        acessa(p,&x1,&y1);

        printf("\n x1=%.2f",x1);
        printf("\n x1=%.2f",y1);
        printf("\n distancia=%f",distancia(p,p1));

        libera(p);
        libera(p1);
        return 0;
}
```

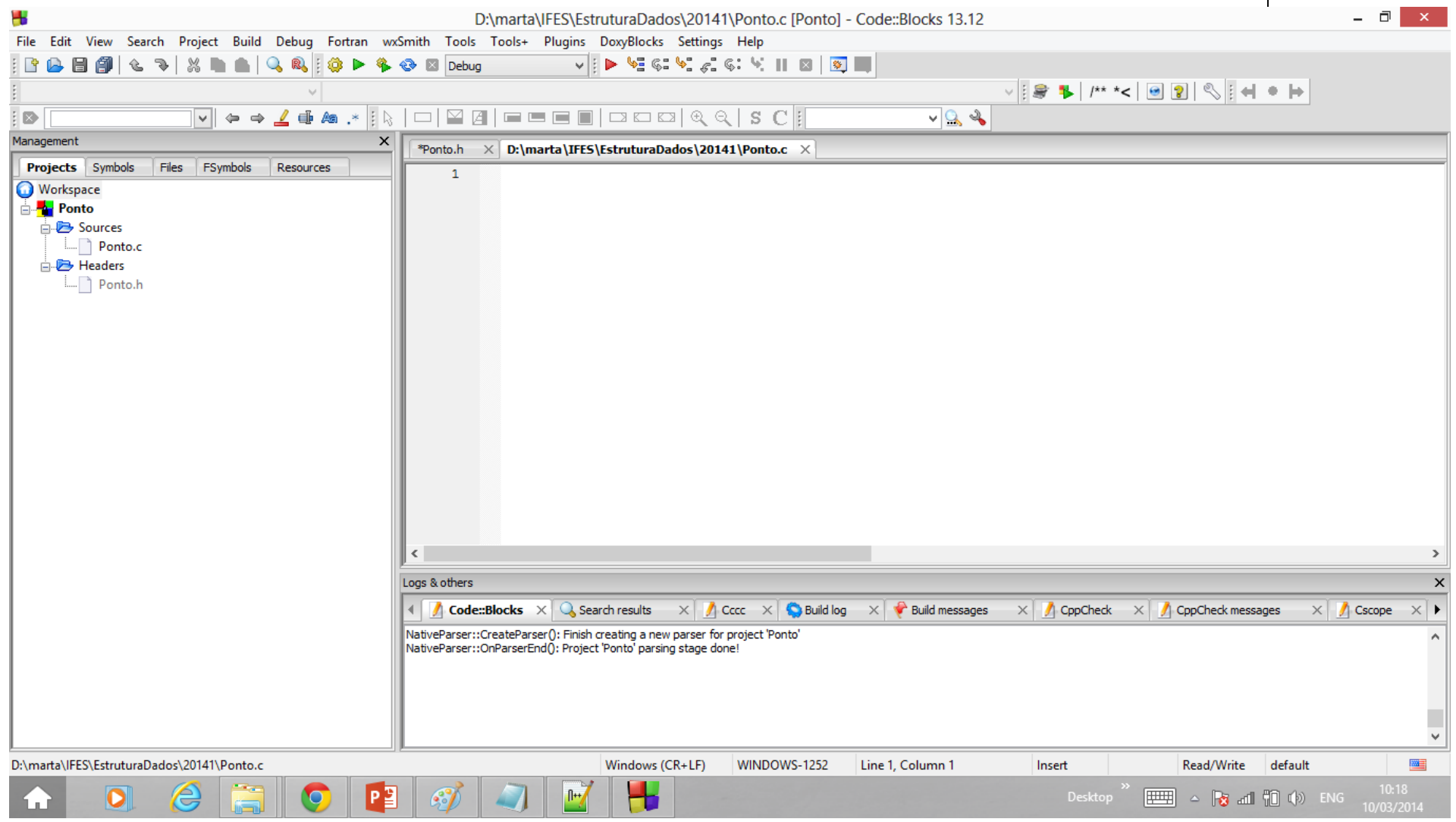
compilar



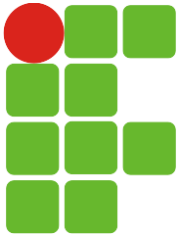
- LINHA DE COMANDO:

>> gcc ponto.c pontoteste.c –o prog

Codeblocks


D:\marta\IFES\EstruturaDados\20141\Ponto.c [Ponto] - Code::Blocks 13.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
Management
Projects Symbols Files FSymbols Resources
Workspace
Ponto
Sources
Ponto.c
Headers
Ponto.h
*Ponto.h
D:\marta\IFES\EstruturaDados\20141\Ponto.c
1
Logs & others
Code::Blocks Search results Cccc Build log Build messages CppCheck CppCheck messages Cscope
NativeParser::CreateParser(): Finish creating a new parser for project 'Ponto'
NativeParser::OnParserEnd(): Project 'Ponto' parsing stage done!
D:\marta\IFES\EstruturaDados\20141\Ponto.c
Windows (CR+LF)
WINDOWS-1252
Line 1, Column 1
Insert
Read/Write
default
Desktop
ENG
10:18
10/03/2014

TAD - resumo final



- Quando fazemos programas pequenos o uso de vários módulos talvez possa não se justificar.

- Para programas médios e grandes, a divisão em módulos é uma técnica fundamental, pois facilita a divisão de uma tarefa maior e mais complexa em tarefas menores, e mais fáceis de implementar e testar. Lema: “Dividir para conquistar”

- Além disso, reutilizar módulos poupa tempo de programação.
- **TODA LINGUAGEM PODEMOS USAR O TAD?**

Exercício no laboratório

Um número racional pode ser expresso como o quociente de dois inteiros a/b.

- Implemente um TAD que contenha as operações:

soma,subtração, multiplicação,divisão

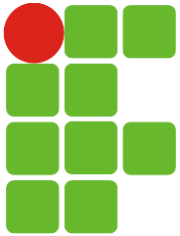
- Implemente um programa principal testando as operações.

$$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \times \frac{d}{c} = \frac{ad}{bc}$$

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

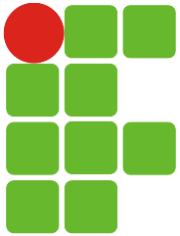
$$\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$$

Exercício no laboratório



```
//criar um número racional: x – numerador / y - denominador
racional set(int x, int y);

//soma dois racionais
racional add(racional x, racional y);

//subtrai dois racionais
racional sub(racional x, racional y);

//Multiplica
racional mul(racional x,racional y);
```

Exercício no laboratório



racional divide(racional x, racional y);

//imprime: sprintf(s, "%i/%i", getN(x),getD(x));

char* notation(racional y);

=> OLHE NO MOODLE A LISTA DE EXERCÍCIOS DISPONÍVEIS.